Universidad Politécnica de Cataluña

Memoria

IA para control de colectivos y NPCs aplicado a un videojuego

Autor: Victor Hugo Rivero
Director: Javier Bejar

9 de Abril, 2018

Contenidos

Resumen

Los videojuegos resultan atractivos a un público muy diverso, y por motivos variados. Y es que son un trabajo que requiere consideración tanto en el apartado visual, como el sonoro, y el diseño de las mecánicas del juego, que terminan siendo una experiencia muy completa para las persona, y un ámbito interesante de estudiar. Los videojuegos son también problemas interesantes de resolver.

El desarrollo de un videojuego está lleno de decisiones, desde sobre de qué tratará el juego, cómo se jugará, como estructurar el código, que algoritmos o técnicas usar, hasta decisiones más detallistas sobre el juego, si usar un sistema o poner un objeto en especial. Además están decisiones de otros apartados, la música, los sonidos, los gráficos, los que suelen ser llevados por artistas, músicos, diseñadores de sonido, entre gente especializada en lograr la mejor expresión del juego. Un videojuego además tiene en cuenta como presenta la información al jugador. Es importante que el jugador entienda de manera más clara y rápida la información que se le da, ya que la interacción que tiene con el problema, que es el juego, depende de su percepción.

La inteligencia artificial también toma decisiones, y aplicada a videojuegos de diversas formas, como para controlar entidades dentro del juego o incluso generar contenido proceduralmente. Claro está que el tipo de inteligencia artificial a utilizar es una decisión más en el desarrollo.

En este proyecto se desarrollará un juego de acción en el que el jugador controla un colectivo, y encontrar qué técnicas de IA se adaptan mejor a la intención del juego.

Abstract

By various reasons, video games turn out to be entertaining to a very diverse public. Since it is a work that requires so much consideration into the visuals, the audio and game design, it comes off as a very complete experience for people, and a interesting field of study.

Video game development is full of choices, from the theme of the game, the gameplay, how to implement the code, to more especific issues like using an especific system or the implementation of an object or entity. Besides there are more considerations into things like music, and visuals of the game, which are taken by musicians or artists, people specialized in their field. Video games also need to take seriously the way they communicate the information to the player. It is important the player understand what is happening in the fastest way possible the information given, since the interaction and feedback to the game depends on it.

Artificial intelligence also is in charge of taking decisions, and in various ways, like controlling entities or generating procedural content. There are different techniques from where to choose too.

In this project, an action game in which the player controls a group of entities, and adapt AI techniques which fit better the game.

Glosario

Este proyecto trabaja con definiciones y términos usados en videojuegos e Inteligencia Artificial. A continuación, se hará un pequeño apartado con una breve definición de palabras relacionadas con este proyecto:

IA Corto para Inteligencia Artificial. Generalmente son algoritmos usados para tomar decisiones. Tienen la característica de que no dan la solución óptima siempre, pero garantizan una calidad a cambio de una ejecución poco costosa, más rapida. Más adelante se explicará tipos de técnicas candidatos a ser usados.

NPC Del inglés *Non-Playable-Character*. Son todos los personajes de un juego que el jugador no controla.

HUD Del inglés *Head-up Display*. Es el método por el cual se transmite información (puntos de salud, puntuación, tiempo restante, etc) al jugador en un videojuego.

UI Del inglés *User Interface*. Se refiere a lo que concierne a la interacción entre el humano y el ordenador.

Colectivo En el contexto de este proyecto, un colectivo es un conjunto de personajes. Este proyecto pretende diseñar un sistema en el cual se controle un colectivo, en vez de un solo personaje como suele ser en la mayoría de juegos. El control de colectivos existe en juegos de estrategia pero no suele estar orientado al combate.

Utilidad Este término dentro del ámbito de la inteligencia artificial se usa para describir a la forma de cuantificar la calidad de una solución encontrada mediante un algoritmo de inteligencia artificial.

Agente En inteligencia artificial, es una entidad que es capaz de percibir su entorno, procesar lo percibido y responder de manera correcta.

Capítulo 1

Definición del Alcance y Contextualización

1.1 Contexto

Actualmente existen muchas herramientas y opciones para lograr el funcionamiento satisfactorio del juego, ya sean plataformas de desarrollo como *Unity*, Godot o Unreal Engine, librerías como SFML, SDL, Allegro, técnicas de IA como Árboles de Comportamiento, Redes Neuronales, etc, que brindan comportamientos y potenciales diferentes para el juego.

Pero también están las herramientas para otros apartados como Blender y Maya para gráficos 3D, Photoshop y Paint Tools SAI para gráficos 2D, FL Studio o Ableton para el apartado sonoro del juego, por mencionar un par de herramientas.

El control de colectivos se ve en los juegos de estrategia. El considerado el primero de los ancestros de este género es el *Utopia (1981)* [3], y carecía de entidades que controlar. La aparición de colectivos comienza con *Nether Earth (1987)* [3]. Haciendo un salto al presente, pasando por los éxitos de Blizzard hasta juegos como *Frozen Synapse*, el control de muchas entidades siempre ha sido algo de los juegos de estrategia. En el terreno de juegos de acción, *Formata* intenta este control de colectivos a una mezcla de estrategia y acción.

1.1.1 Actores Implicados

Desarrollador

Yo, el autor de este proyecto, desarrollaré el proyecto. Al tratarse de un videojuego que necesita material audiovisual, también haré el papel de artista, diseñador, músico, etc, si hace falta.

Director del proyecto

El director de este proyecto es Javier Béjar, quien se encarga de supervisar el desarrollo del proyecto. También será asistente y guía, aportando la experiencia sobre Inteligencia Artificial.

Testers

Se consultará a personas escogidas por el desarrollador, para que den opiniones, valoraciones y pongan a prueba el proyecto, sin conocimiento técnico del mismo. Un videojuego es valorado subjetivamente a la hora de decir si es divertido o interesante. Por eso es importante la opinión de testers.

Usuarios Finales

Una vez se obtenga un producto terminado, será expuesto a un público con el fin de ser usado. Este público son los usuarios finales.

1.2 Estado del Arte

1.2.1 Técnicas de IA

A continuación se explicarán las técnicas principales[2] que se tienen en cuenta para el proyecto a realizar. En la realidad existen más técnicas, pero no forman parte del alcance del proyecto.

Comportamiento Ad-hoc

Este clase de métodos se basa en definir un comportamiento a base de reglas. Suelen ser las más simples de implementar y las más conocidas, pero carecen de flexibilidad, en el sentido que definen un comportamiento estático que no aprende ni evoluciona. Dentro de esta categoría tenemos los siguientes métodos:

FSM (Maquina de estados) Un FSM esta compuesto por 3 componentes básicos: estados, transiciones y acciones. Los estados representan y guardan información sobre el estado o tarea que se esta llevando en ese momento. Las transiciones indican las condiciones que se han de cumplir para cambiar de estado, y de que estado a cual se cambia. Las acciones son aquello que se hace en cada estado. Este método es implementado con un grafo en el que cada nodo es un estado y las transiciones son las aristas.

BT (Árboles de comportamiento) Un BT logra modelar una inteligencia mediante comportamientos, asi como una FSM lo hacia con los Estados. Esto lo logra empleando una estructura de árbol, y navegandolo de padre a hijos. La forma de navegar un árbol depende del tipo del nodo padre: un nodo de secuencia navega todos los hijos en un orden, tiene éxito si todos los hijos tienen éxito y falla si alguno falla; un nodo seleccionador

escoge alguno de sus hijos, si escoge por prioridad ejecuta sus hijos en un orden hasta que alguno tenga éxito y falla caso contrario, si escoge por probabilidad tiene éxito si el hijo seleccionado tiene éxito y falla caso contrario.

IA basado en utilidad Esta técnica consiste en usar una funciones de utilidad para tomar decisiones. La utilidad se usa para valorar las opciones a tomar por el agente.

Técnicas evolutivas

Las técnicas evolutivas son algoritmos de busqueda que combinando dos soluciones se intenta encontrar una solución aún mejor.

El primer aspecto a tener en cuenta es la codificación del problema, ya que pueden haber buenas y malas codificaciones, en el sentido que facilitan o impiden encontrar una buena solución.

Una vez codificado el problema se suele seguir el siguiente esquema:

Inicialización Se comienza el problema con N soluciones creadas aleatoriamente.

Evaluación Se evalua el fitness de cada solución, que es como se le llama a la utilidad en los algoritmos evolutivos.

Selección de padres Se seleccionan los padres de las nuevas soluciones, generalmente usando el fitness.

Reproducción Se generan nuevas soluciones usando los padres, generalmente combinandolos.

Mutación De las nueva población de soluciones, se mutan algunos, es decir, se cambia aleatoriamente alguna parte de la codificación de la solución.

Reemplazo Se descartan algunas soluciones, con algún criterio. Algunos criterios son, por ejemplo, conservar los que tengan mejor fitness, o descartar todos los padres.

Terminación En caso de obtener una solución deseada, se termina, si no se vuelve a ejecutar otra iteración del algoritmo.

Aprendizaje Supervisado

Este tipo de método consiste en entrenar una inteligencia, dando ejemplos (datos etiquetados), para que sea capaz de reconocer datos sin etiquetar. Un factor a tomar en cuenta es que datos se extrae para cada ejemplo, por ejemplo, si quieres clasificar frutas, para toda las frutas querrás datos como medida y el color, pero cosas como la fecha esperada de descomposición no interesan o aportan ruido a la clasificación. Algunos métodos de aprendizaje supervisado son:

ANN (Redes Neuronales) Inicialmente se tuvo como un modelo del cerebro humano. La unidad básica de una red neuronal son las neuronas. Una neurona recibe varias entradas de otras neuronas, cada entrada multiplicada por un peso (representado por las conexiones), dentro de la neurona que recibe se suman las entradas y posiblemente se sume un bias, luego se aplica una función de activación (que es una función matemática), y el resultado se vuelve la salida de la neurona. Una red neuronal es simplemente una conexión de muchas neuronas en las que entran datos por la primera capa de neuronas, y tras capas de neuronas se obtiene una salida por cada neurona final. La forma de usar una red neuronal para obtener resultados es ajustando los pesos de las conexiones y de los posibles bias, mediante una técnica llamada backpropagation, el cual no se explicará porque escapa al motivo del proyecto.

DT (Árboles de decisiones) Este método usa la estructura de un árbol. La entrada es representada por los nodos del árbol y la salida son las hojas del árbol. El árbol es generado cogiendo un atributo de los ejemplos y usándolo para dividir lo mejor posible los datos etiquetados. Para esto

1.2.2 Videojuegos Relacionados

Muchos métodos y adaptaciones de IA son usadas en los videojuegos actuales, sin embargo algunos juegos que no son necesariamente modernos han marcado o hecho populares su uso en los videojuegos, técnicas que se siguen usando hasta ahora. Algunos de estos juegos son[1]:

Thief: Deadly Shadows Este juego del 2004 uso un sistema sensorial que permite a los personajes reaccionar a su entorno. Este sistema es explotado dado el hecho que es un juego de sigilo.

Creatures Este juego usa Redes Neuronales para que las criaturas aprendan a sobrevivir.

Halo Franquicia conocida por volver popular el uso de árboles de comportamiento, especialmente desde Halo 2.

F.E.A.R Produce comportamientos que tienen en cuenta el contexto mediante el uso de un panificador.

Black and White Un conjunto de técnicas es usado en el mismo juego, como BT y ANN.

1.2.3 Plataformas de Desarrollo

El aumento de complejidad ante usar librerías y la rápida agilización de la industria de los videojuegos, produjo la creación de IDE (Integrated Development Enviroment), programas que son un conjunto de herramientas para el desarrollo. Plataformas especializadas para el desarrollo de videojuegos fueron

creadas, todas con alguna característica o motivos específico. En este proyecto usaremos *Unity*, una plataforma de desarrollo que es usado en la industria para el prototipado de videojuegos e incluso para la elaboración de productos finales. Pero existen otras plataformas más como Game Maker, especializado en 2D y caracterizado por un lenguaje propio (Game Maker Language) además de un método de programación por bloques, y Godot, una nueva plataforma con mejores avances en el apartado 2D y un sistema de Nodos para el control de entidades en el juego. Estas plataformas pretenden agilizar y facilitar el flujo de trabajo, integrando sistemas que de otra forma debería implementar el desarrollador.

1.2.4 Conclusiones

Para este proyecto, el conocimiento que se abarca comprende solo el manejo de la herramienta de desarrollo, que como ha sido mencionado es *Unity*, y las técnicas de IA disponibles, de las cúales se han mencionado unas cuantas que servirán de base para la IA diseñada para este proyecto, sin embargo lo más probable es que no se prueben todas, si no las más básicas, y apartir de ahi se desarrolle. La parte gráfica y de sonido no se han mencionado, ya que no son percibidos como un problema al que se van añadiendo mejores soluciones. Bien es cierto que mejoras en el apartado gráfico han ocurrido y que ahora se tienen más herramientas para estos apartados, pero el que un juego use menos recursos de gráficos o de sonido no representa una limitación, sino una decisión de diseño.

1.3 Objetivos

El problema al que intenta dar respuesta este proyecto hace referencia al diseño de juegos, implementación de conocimientos informáticos y diseño gráfico. Primero ingeniar ideas para un videojuego, y hacer la selección de que ideas funcionarían mejor, como hacerlas realidad e implementarlas, que herramientas usar, etc. Todas aquellas decisiones que se hayan de tomar en el desarrollo de un videojuego forman parte del problema que se presenta en este proyecto.

Este proyecto busca, mediante el desarrollo de un videojuego, los siguientes objetivos:

- Adaptar, valorar e implementar técnicas de IA en una aplicación real; en este caso, un videojuego. Se buscará que tipo de IA es mejor para los diferentes tipos de NPCs y otras entidades en el juego.
- Diseñar e implementar una interfaz gráfica efectiva al comunicar información al jugador. Esto comprende lo que es la HUD, UI, y demás apartado gráfico.
- Diseñar y ajustar la parte de control e interacción con el usuario.

- Finalmente, aplicar la estrategias algorítmicas necesarias para resolver problemas reales del desarrollo de la aplicación.

1.4 Alcance

1.4.1 Definición

Para poner en práctica la elaboración de un programa que interactúa con un humano como lo es un videojuego, y la inteligencia artificial que utiliza, se diseñará y desarrollará un videojuego.

Una vez implementado la parte básica del juego, se implementarán las inteligencias artificiales, para cada tipo de entidad posible en el juego.

Cuando se tengan IAs funcionales se pondrán a prueba, y se terminará de implementar lo que quede del juego planeado. Esto comprende inteligencia artificial a base de reglas.

Finalmente se testeará con el objetivo de encontrar bugs y de validar que se ha conseguido el comportamiento correcto de la IA.

1.4.2 Obstáculos

Calendario (Planeamiento)

El plazo de tiempo del que se consta para el proyecto son 4 meses, que es compartido con otras responsabilidades. Para superar este obstáculo, el proyecto constará de una parte básica indispensable y una parte opcional, con prioridades diferentes. Se llevará un control guiado por un planeamiento previamente hecho.

Infactibilidad

Puede ser que opciones que no sean factibles porque la plataforma de desarrollo no lo soporta o porque el tiempo esperado de implementación no encaja con la planificación. Para prevenir esto, éstas opciones serán descartadas o se les asignará una prioridad más baja lo más pronto que se pueda. Antes de empezar una tarea del proyecto, se evaluarán las opciones y se empezarán con la más simples, en caso de no ser posible alguna opción descartarla o de no saber cómo ejecutarla dejarla para el final. Usar la planificación para saber si se está detrás de lo planeado.

Bugs

Comportamientos erróneos y mal funcionamiento de código puede ocurrir a lo largo del proyecto. Por esto se realizarán pruebas a menudo y se verificará que el programa hace lo que presuntamente hace.

1.5 Metodología

1.5.1 Herramientas

Se usará *Unity*, que es una plataforma de desarrollo de videojuegos. Usar librerías para un lenguaje de programación querría decir planificar e implementar una variedad de sistemas que no está relacionado al proyecto (Como el manejo de escenas, interfaz gráfica, etc), que además requerirían de tiempo que se puede invertir en esfuerzos por lograr los objetivos del proyecto. De entre las posibles plataformas a usar, como Godot o Game Maker, se ha escogido *Unity* por su comunidad activa, su uso en la industria tanto para prototipado de juegos como para elaboración de productos finales, su documentación y las posibilidades que ofrece.

1.5.2 Validación

Para validar la eficacia del juego en la respuesta del control y en el informar al jugador lo que ocurre en el juego, se usará la valoración de testers y el control del director. La eficacia de la IA dentro del juego depende de si lo hacen divertido (valorado por testers), si logran los requerimientos básicos para los que están hechos, y si aportan algún comportamiento interesante.

1.6 Cambios en el Alcance y Contextualización

En esta sección no han habido cambios. El proyecto sigue cumpliendo con el alcance definido y los objetivos propuestos. Por lo tanto, los obstáculos analizados también son los mismos. La metodología del proyecto sigue siendo la misma planteada desde un inicio, así como la herramienta principal, Unity.

Capítulo 2

Planificación Temporal

2.1 Plan de proyecto

El tiempo de desarrollo del proyecto es de 4 meses aproximadamente, desde el 19 de febrero hasta el 29 de junio. Se procederá a abstraer tareas y objetivos y a identificar dependencias entre ellas.

2.1.1 Hito inicial (Planificación)

Esta es la fase actual del proyecto, en la que se planifica el proyecto y se describen las partes del mismo. Esta parte consiste en lo siguiente:

- Definición de Alcance y Contextualización
- Planificación Temporal
- Gestión económica y sostenibilidad

2.1.2 Análisis y Diseño

En esta fase se describirá, diseñará especificamente como funciona el proyecto, en este caso, el videojuego; y se tomarán las medidas y decisiones iniciales del proyecto, lo que quiere decir, plasmar las ideas y opciones posibles y validar su posibilidad dentro del proyecto.

Se concretará la mecánica del videojuego y como se lleva a cabo dado la plataforma de desarrollo escogida. Esto es cuáles son los elementos del juego, como interactúan entre ellos, como es el entorno en el que interactúan, que objetivo tiene el jugador, etc.

Se evaluarán las opciones de inteligencia artificial y como llevarlas a cabo en la plataforma de desarrollo escogida. Se tendrán en cuenta las opciones y según

la factibilidad y el comportamiento esperado, se escogerá usar algunas en específico.

Se tendrá en cuenta también el apartado de la interfaz gráfica. Se diseñará la parte gráfica con el objetivo de facilitar y acelerar la comprensión del jugador sobre los eventos del juego.

2.1.3 Descripción de las tareas

Debido al uso de una metodología agil y flexible, seguir un proceso estricto de tareas no es realista ni pragmático dada la naturaleza experimental del proyecto. Sin embargo, se pueden distinguir ciertas etapas y plantear una duración para cada una con el objetivo de guiar el desarrollo del proyecto y tener un estimado en todo momento del progreso del mismo.

En el diagrama de Gantt se ha intentado marcar un proceso de tarea tras otra. No obstante, hay tareas que tienen el mismo plazo porque llevan la misma prioridad, cosa que puede cambiar en el transcurso del proyecto. También hay tareas que van detrás de otras y son disjuntas, a pesar de no tener dependencia. Esto se hace por orden de prioridad y por evitar tener muchas tareas a la vez.

Algunas tareas estan descompuestas en tareas más pequeñas, otras no. Esto es por evitar aumentar detalle innecesario a al planificación y apegarse más a lo que realmente será.

Al llevar a cabo el proyecto, la ejecución de las tareas esta sujeto a variaciones tanto en orden como en duración.

Las tareas de las que consta el proyecto se enumeran de la siguiente forma:

1. Hito Inicial
2. Analisis y Diseño
3. Desarrollo de los actores
4. Desarrollo de IA
5. Desarrollo del escenario
6. Desarrollo del audiovisual
7. Pruebas y ajustes
8. Hito Final

El desarrollo de los actores consiste la implementación de los personajes controlados por el jugador y los NPC. El desarrollo de la IA es la implementación

del comportamiento de estos actores. El desarrollo del escenario es la implementación del entorno en el que los actores interactúan.

Estas tareas son levadas por el desarrollador.

La música y lo gráfico esta bajo una sola tarea, que será llevada por el músico y el artista.

Las pruebas y los ajustes finales serán realizados por el desarrollador y consiste en ajustar y corregir funcionalidades del juego.

2.1.4 Hito final

En esta fase se recolectará y entregará todo recurso usado para llevar a cabo el proyecto (imagenes, audio, codigo, etc). Se elaborará una memoria y demás documentación necesaria para la entrega final. También se incluirá el juego en si, una build lista para ser ejecutada y probada.

2.1.5 Valoración de alternativas

Esta planificación esta pensada para ser flexible y adaptarse, de ser necesario se dedicará más tiempo a las tareas que lo demanden. Se tratará de mantener el ritmo que marca la planificación. Y en el peor de los casos se simplificará o excluirá aquello que no sea necesario o prescindible. Toda solución o decisión frente a un problema constará en la memoria final. A continuación se pretende analizar brevemente posibles escenarios que retrasen el desarrollo del proyecto:

Problema de diseño

Puede ocurrir que al llevar acabo alguna funcionalidad o implementar una parte del juego no funcione como se tenía previsto, o que existan decisiones de diseño que no van bien juntas. En cualquier caso, habrá que dedicar tiempo a replantear parte del proyecto o de solucionarlo de alguna forma en la implementación. Este tipo de atrasos se pretende evitar con un diseño básico y aumentar el volumen a partir de algo ya valorado y validado.

Tiempo de implementación mayor al previsto

A la hora de implementar algo puede ocurrir que sea totalmente factible pero por algún motivo requiere de un tiempo mayor al previsto, por ejemplo el entrenar redes neuronales o que no se puede validar algo implementado porque dependa de algo más. Se reaccionará ante estos problemas valorando dos tipos de soluciones, terminar de implementar lo que se esté desarrollando y planear de que posible tarea extraer tiempo para compensar, o dejar como tarea incompleta para terminar al final si alcanzan recursos.

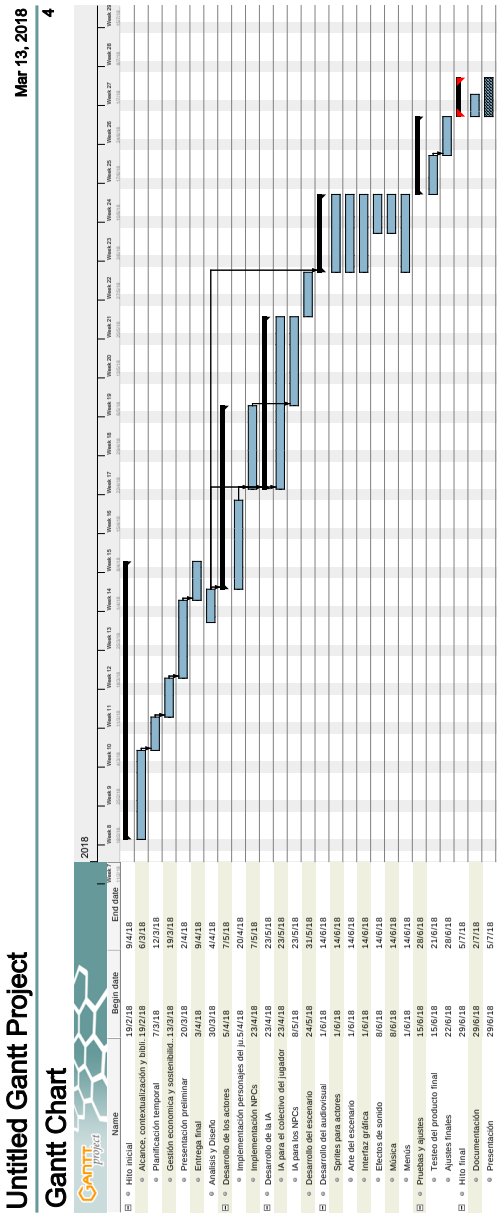
Funcionalidad de implementación difícil

La falta de experiencia combinado al poco tiempo de desarrollo o las limitaciones de las herramientas puede dar a lugar que alguna funcionalidad sea difícil de implementar. Es por eso que el proyecto parte de una base simple. Esto puede ocurrir en fases más posteriores, pero se garantizará que lo se tiene hecho antes es más sencillo de implementar.

Funcionamiento incorrecto

Puede que al implementarse algo se haya hecho de forma incorrecta o que al implementarse otro sistema interactúe de forma no esperada ni deseada. Estos problemas ocurren sobre trabajo ya realizado por lo que significa que se ha de corregir.

2.2 Diagrama de Gantt



2.3 Duración aproximada

Tarea	Duración
Hito Inicial	160
Analisis y Diseño	20
Desarrollo de los actores	100
Desarrollo de IA	110
Desarrollo del escenario	30
Desarrollo del audiovisual	40
Pruebas y ajustes	40
Hito Final	10
Total	510

2.4 Recursos

2.4.1 *Software*

- *Unity*
- Windows 10
- GIMP
- Google Docs
- OpenMPT (posibilidad)
- Famitracker (posibilidad)

2.4.2 *Hardware*

- Periféricos
- Ordenador

2.4.3 Recursos humanos

- Director del proyecto
- Desarrollador
- Artista
- Músico

2.5 Cambios en la Planificación Temporal

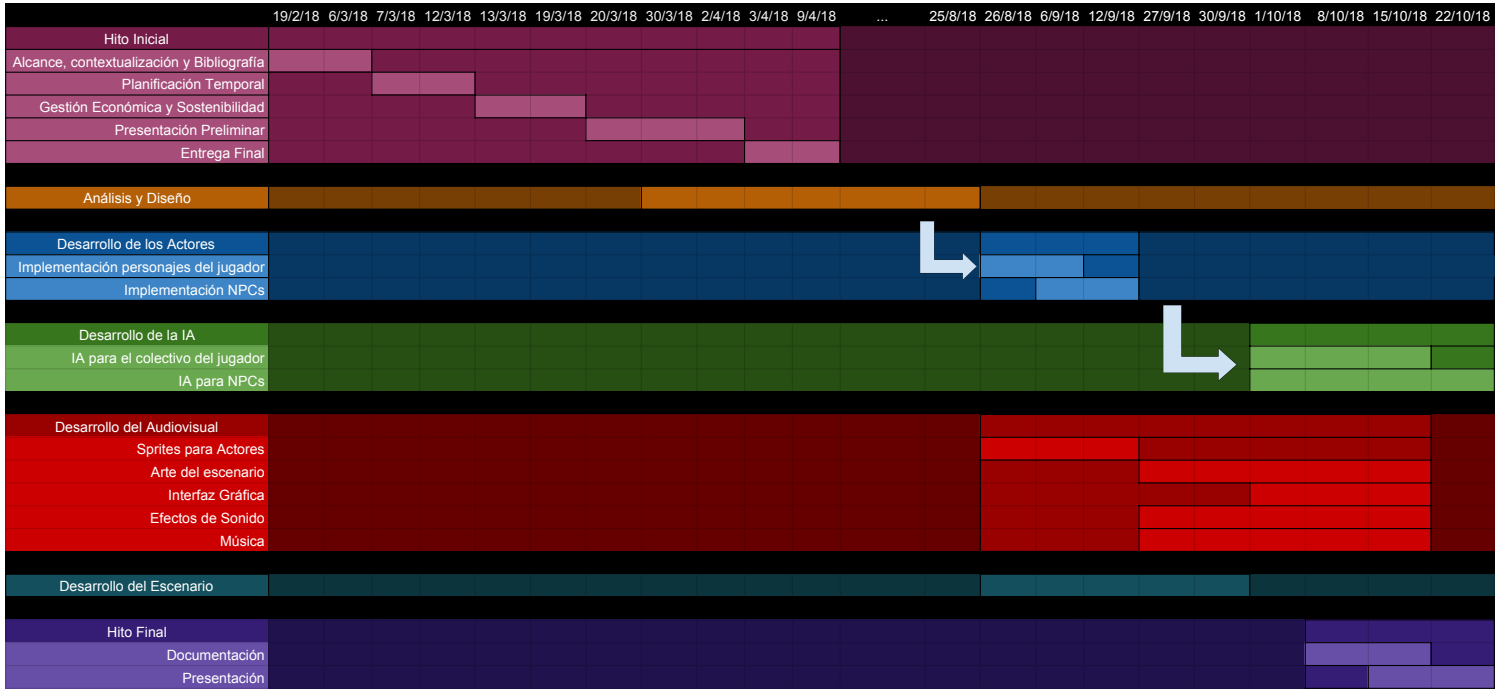
En este apartado han habido una serie de cambios. La división de tareas y la descripción de cada una son suficientemente genéricas para no necesitar ser cambiadas o adecuadas a arreglos concretos del proyecto. Sin embargo, la planificación del Gantt y la distribución en el tiempo han cambiado. A continuación los cambios comentados:

2.6 Diagrama de Gantt

El Gantt ha cambiado principalmente por la extensión del TFG. Durante el verano se prolongó la fase de analisis y diseño, con el motivo de aprender y refinar el uso de Unity, la herramienta principal usada en el proyecto.

Las dependencias marcadas son la que existe entre el analisis y diseño con el resto del trabajo, en este caso la lógica del juego que es lo primero que se hace; y la dependencia entre la lógica de los actores y su IA.

Hay varias casillas que están como llevadas a cabo simultáneamente, esto es porque o han sido llevadas simultaneamente o porque no hay ninguna diferencia en hacerlas en cualquier orden.


19/2/18
6/3/18
7/3/18
12/3/18
13/3/18
19/3/18
20/3/18
30/3/18
2/4/18
3/4/18
9/4/18
...
25/8/18
26/8/18
6/9/18
12/9/18
27/9/18
30/9/18
1/10/18
8/10/18
15/10/18
22/10/18
Hito Inicial
Alcance, contextualización y Bibliografía
Planificación Temporal
Gestión Económica y Sostenibilidad
Presentación Preliminar
Entrega Final
Análisis y Diseño
Desarrollo de los Actores
Implementación personajes del jugador
Implementación NPCs
Desarrollo de la IA
IA para el colectivo del jugador
IA para NPCs
Desarrollo del Audiovisual
Sprites para Actores
Arte del escenario
Interfaz Gráfica
Efectos de Sonido
Música
Desarrollo del Escenario
Hito Final
Documentación
Presentación

2.7 Duración aproximada

Tarea	Duración
Hito Inicial	160
Analisis y Diseño	45
Desarrollo de los actores	45
Desarrollo de IA	120
Desarrollo del escenario	25
Desarrollo del audiovisual	45
Hito Final	10
Total	450

Aquí ha habido una distribución de horas diferente. Se ha disminuido el desarrollo de los actores en gran magnitud, ya que se ha terminado ya y no ha llevado tanto tiempo. Se ha eliminado la fase de pruebas ya que se ha ido probando al finalizar cada fase el correcto funcionamiento del trabajo. Se ha aumentado un poco la fase de IA y audiovisual, que son las que faltan por terminar, dando así un margen mayor.

2.8 Recursos

2.8.1 *Software*

- *Unity*
- Windows 10
- GIMP
- Google Docs
- Bosca Ceoil
- Panda BT (Unity)

2.8.2 *Hardware*

- Ordenador

2.8.3 Recursos humanos

- Director del proyecto
- Desarrollador
- Artista
- Músico

Sobre los recursos, han habido algunos pequeños cambios. Para empezar, se ha decidido Bosca Ceoil como software para audio y música; también se lista la librería usada para la IA de los NPCs. Se han quitado los periféricos del hardware, por ser innecesario a los objetivos del tfg, además de poder suponer problemas al tener que dedicarle un esfuerzo extra al manejar el input.

Capítulo 3

Gestión Económica y Sostenibilidad

3.1 Autoevaluación de sostenibilidad

La formación que tengo consta de solo alguna charla que se ha dado sobre sostenibilidad en la universidad y las lecturas y el trabajo autónomo que realizo en esta asignatura. Por lo tanto, en cuanto a sostenibilidad y análisis estoy solo formado en lo básico. No soy consciente de conocer técnicas, tecnologias o estrategias para aumentar la sostenibilidad de un proyecto, por lo que reconocer o proponer algo de este estilo me resulta díficil si no infactible.

Mi inexperiencia tanto con el aspecto de sostenibilidad en proyectos como gestionando un proyecto también juega un papel. Me cuesta entender ciertos aspectos económicos y prever riesgos y controlar la parte económica y de planificación, calcular costos y amortizaciones lo puedo hacer a un nivel poco detallado.

Sin embargo, la problemática social y ambiental relacionada con las TIC no me es ajeno, a pesar de la falta de experiencia y formación. Soy capaz de usar indicadores para medir el impacto de un proyecto en el aspecto social y ambiental, de forma básica. También trato de maximizar el impacto positivo que pueda tener un proyecto sobre la sociedad y sobre el ambiente. Tengo en cuenta temas como la seguridad, ergonomía asi como la justicia social, la transparencia y la igualdad. En cuanto al trabajo colaborativo, he tenido experiencia llevando un mismo proyecto entre un equipo, y soy capaz de usar herramientas y de evaluar las implicaciones del trabajo colaborativo.

Finalmente, poco sé sobre los principios deontológicos, por lo menos de manera consciente. Por lo tanto, tampoco sé como se relaciona con la sostenibilidad dentro de un proyecto y mucho menos proponer soluciones teniendolos en cuenta voluntariamente.

3.2 Presupuesto

3.2.1 Recursos Humanos

En este proyecto todos los roles los desempeñara una sola persona. A continuación se definirán los roles del proyecto, y se elaborará un presupuesto para los trabajadores.

Rol	Horas	Precio por hora (en euros)	Precio total
Director de Proyecto	170	50	8500
Desarrollador	210	35	9100
Músico	20	15	300
Artista	20	15	300
Total	510		18200

3.2.2 *Hardware*

Para llevar a cabo el proyecto se usarán los recursos señalados ya en la planificación: un portatil y periféricos, es decir, mandos o joysticks.

Producto	Precio (en euros)	Unidades	Vida Útil (en años)	Amortización
HP ProBook 6470b Notebook	300	1	5	25
Periféricos	10	2	3	1,7
Total	320			26,7

3.2.3 *Software*

Todo el *software* usado en este proyecto es gratuito. GIMP, OpenMPT, Famitracker y Google Docs son gratuitos, y no hay versión de pago. Sin embargo *Unity* tiene una versión de pago, y Windows 10 es pagado, de no ser por la licencia de estudiante que nos da la universidad.

Producto	Precio (en euros)	Vida Útil (en meses)	Amortización
Windows 10	0	-	-
Unity	0	-	-
GIMP	0	-	-
OpenMPT	0	-	-
Famitracker	0	-	-
Google Docs	0	-	-
Total	0		-

3.2.4 Costes Indirectos

Los siguientes costes son los causados por el proyecto que no están considerados en el resto de categorías.

Producto	Precio (euros por unidad)	Unidades	Precio Estimado
Electricidad	0,07	2000 kWh	140
ADSL	35	4 meses	140
Total			280

3.2.5 Presupuesto total

Aqui se suman los presupuestos de cada apartado y se añade una cantidad de contingencia, el cual se ha escogido que sea un 10%. Se han valorado posibles escenarios que pueden retrasar el proyecto y a pesar de que se han dado soluciones, no se descarta que puedan ocurrir estos u otros escenarios, por esto la contingencia.

Coste	Presupuesto
Recursos Humanos	18200
Software	0
Hardware	346,7
Indirectos	280
Total	18826.7
Total + Contingencia (+10%)	20709.37

3.2.6 Asignación de costos a tareas

En la siguiente tabla se evalúa el coste de cada actividad y el rol que la desempeña.

Tarea	Coste (en euros)	Rol
Hito Inicial	8000	Director
Analisis y Diseño	700	Desarrollador
Desarrollo de los actores	3500	Desarrollador
Desarrollo de IA	3850	Desarrollador
Desarrollo del escenario	1050	Desarrollador
Desarrollo del audiovisual	600	Músico / Artista
Pruebas y ajustes	1400	Desarrollador
Hito Final	500	Director

3.3 Control de Gestión

Las desviaciones pueden ocurrir las etapas de desarrollo llevadas por el desarrollador, es decir, el desarrollo de los actores, la IA y el escenario. Tal como esta diseñado, en el peor de los casos, se puede prescindir de un escenario elaborado y así recortar tiempo de la tarea que se le dedica. La etapa de Pruebas y Ajustes esta para, de cierta forma, amortiguar el impacto temporal que se pueda sufrir.

Para medir el desvio se usará el consumo en horas de las tareas, usando lo esperado y el consumo real. De esta forma, será necesario llevar un registro de

horas trabajadas. Obviamente la diferencia en el consumo significa también una desviación en el coste del proyecto.

El plan entonces es aumentar las horas de trabajo en la fase final de Pruebas y Ajustes a la cual se le dedica 40 horas, y dura 1 semana aproximadamente. 2 horas por dia extra durante toda la semana se traduce en 350 euros extra para un total de 201239.37 euros.

3.4 Análisis de Sostenibilidad

La siguiente tabla contiene la evaluación de la sostenibilidad de este proyecto hasta el momento:

	PPP	Vida Útil	Riesgos
Ambiental	9/10	-	-
Económico	8/10	-	-
Social	9/10		-
Sostenibilidad	26/30	-	-

3.4.1 Dimensión Económica

El presupuesto calculado para este proyecto esta lleno de estimaciones y de calculos basados en supuestos. Además no se han tenido muy en cuenta riesgos y se ha dejado un margen basado en la intuición. Al ser un proyecto con pocos requerimientos en el sentido de que el apartado de *hardware* y *software* no es algo que figure contrastantemente en el presupuesto y lo que realmente pesa que son los recursos humanos son llevados por una persona que no lo hace por recibir un pago, el apartado del presupuesto no ha sido tan detallado como podría haber sido.

En cuanto el estado del arte actual, en este proyecto todo el *software* que se usa es gratis, no representa ningún costo. En contraste a los estudios de videojuegos que puedan usar programas con licencias caras temporales, en este proyecto estos recursos no hacen que el proyecto requiera de mayor presupuesto.

3.4.2 Dimensión Ambiental

Este proyecto no utiliza más recursos de los que necesita. Ambientalmente tiene impacto muy reducido, utiliza *hardware* como un portátil y periféricos que ya se tienen, y el *software* en si no causa ningún impacto negativo. Lo único que puede causar un impacto en este tipo de proyecto es el consumo de electricidad, pero no se usa más de lo que se considera un uso diario. El impacto exacto no ha sido calculado, se ha calculado el coste de la electricidad y el consumo, pero al no ser nada especial no se le ha prestado demasiada atención.

Este proyecto no busca minimizar el impacto ambiental, ni su desarrollo llevará

a alguna forma de reducir el impacto ambiental como consecuencia. Actualmente las soluciones que se usan comparadas a la que se desarrollará tampoco busca mejorar el impacto ambiental, son dos cosas independientes.

3.4.3 Dimensión Social

En el aspecto social, el mayor beneficiario de este proyecto soy yo, el que lleva el proyecto. Me permitirá obtener experiencia en desarrollo de un videojuego, el desarrollo de inteligencia artificial para un videojuego y diseño de una interfaz para el usuario. Me dará la oportunidad de aplicar los conocimientos aprendidos a lo largo de la carrera y, al tener una experiencia más cercana, ayudarme a decidir en que me quiero enfocar a futuro.

El enfoque de este proyecto es en la aplicación y adaptación de técnicas conocidas a un videojuego y el desarrollo del mismo desde 0 usando recursos al alcance. Lo nuevo que surgirá del proyecto será el videojuego y la adaptación de la técnica de inteligencia artificial, sin embargo, esto no prevee ninguna mejora a la calidad de vida de nadie. La necesidad del proyecto es a nivel personal más que una solución a alguna problemática social existente.

3.5 Cambios en la Gestión Económica

Los cambios en la gestión económica son consecuencia de los cambios en la distribución de horas en las tareas y en los recursos.

Estas son las tablas que han cambiado:

3.5.1 Recursos Humanos

Rol	Horas	Precio por hora (en euros)	Precio total
Director de Proyecto	170	50	8500
Desarrollador	235	35	8225
Músico	20	15	300
Artista	25	15	375
Total	510		17400

3.5.2 *Hardware*

Producto	Precio (en euros)	Unidades	Vida Útil (en años)	Amortización
HP ProBook 6470b Notebook	300	1	5	25
Total	300			25

3.5.3 Presupuesto total

Coste	Presupuesto
Recursos Humanos	17400
Software	0
Hardware	325
Indirectos	280
Total	18005
Total + Contingencia (+10%)	19805.5

La table de análisis de sostenibilidad

	PPP	Vida Útil	Riesgos
Ambiental	9/10	15/20	-7/20
Económico	8/10	10/20	-2/20
Social	9/10	8/20	-0/20
Sostenibilidad	26/30	43/60	-9/60

El análisis completo de la tabla:

Vida Útil:

Ambiental El costo ambiental del proyecto durante su vida útil es reducido, el equivalente al uso del ordenador de quien lo use, es decir, gasto de electricidad e indirectamente de la vida del ordenador que se use. Sin embargo, no ayuda a reducir ningún costo ambiental ni evitarlo.

Económico Durante su vida útil no necesita ningún tipo de mantenimiento ni de retoque, pero se puede seguir trabando en el proyecto, mejorando las estrategias usadas o implementando nuevas. Sin embargo esto depende de la intención que se le quiera dar al proyecto, y en este caso no hace falta más.

Social Los colectivos que se benefician más de este proyecto son: gente que juega videojuegos, y programadores que están aprendiendo a desarrollar juegos o usar Unity. Este proyecto sirve tanto como juego como ejemplo para quien quiera aprender. Sin embargo no resuelve ninguna problemática social

Riesgos:

Ambiental Lo que pueden ocasionar los riesgos de este proyecto como máximo, es mayor consumo de electricidad, y en el peor de los casos, gasto del hardware, como el ordenador que es usado.

Económico Podrían producirse escenarios como necesidad de compra de algún software para el desarrollo, o la compra de algún hardware gastado, como el ordenador, y en el caso de los recursos humanos, la necesidad de aumentar el tiempo de trabajo y por lo tanto el salario. Sin embargo, ninguno se ha producido.

Social Este proyecto no perjudica a nadie.

Capítulo 4

Regulaciones

4.1 PEGI

Al tratarse de un videojuego, puede clasificarse con el sistema PEGI (Pan European Game Information). Este sistema es usado en Europa para clasificar contenido por edades (3, 7, 12, 16, 18). Para clasificar el contenido, se usan los siguientes indicadores de contenido:

- Violencia
- Lenguaje soez
- Miedo
- Sexo
- Drogas
- Discriminación
- Apuestas
- Online

En este caso en particular, el juego llevaría solo un indicador de violencia, por el uso de armas o elementos de acción que tiene el juego. Al tener un mínimo de violencia y ningún otro indicador, el mínimo clasificador de edad que se le puede dar es el de 7.

4.2 Licencia de Unity

El juego esta hecho con Unity, un programa que es usado gratis, pero que tiene versiones pagadas y licencias disponibles. En caso que el producto final consecuencia de este proyecto sea comercializado, se ha de tener en cuenta las

herramientas usadas y si se permiten la comercialización de contenido hecho con tales herramientas.

Todo el software usado es gratis y libre (ej. Bosca Ceoil) o pagado (ej. Windows 10). En cuanto a Unity, este permite la comercialización de contenido bajo ciertas condiciones.

Se puede desarrollar y comercializar juegos hechos con Unity mientras se cumplan los requisitos para usar la *Personal Edition* de Unity, esto es, mientras no se hayan ganado / recibido mas de 100,000 dólares en ingresos / fondos en el ultimo año fiscal.

4.3 Cambios en las Regulaciones

Los cambios y ajustes que han habido en otros apartados no han afectado a las regulaciones que rigen este proyecto.

Capítulo 5

Diseño

Para el apartado de Diseño de este proyecto, al tratarse de un videojuego, se ha considerado oportuno crear un GDD (*Game Design Document*). Un GDD es, como su nombre lo indica, el documento donde constan todos los elementos del videojuego y se registra el proceso de diseño del mismo. Este mismo es lo que se conoce como un documento vivo, es decir, que es actualizado regularmente a lo largo de desarrollo.

Este GDD esta dividido en los diferentes apartados que conforman el videojuego:

5.1 Personajes

En este juego existen 5 tipos de entidades cada una con habilidades diferentes:

Cazador Llamada *Hunter* en el código. Su habilidad es atacar con un disparo, el cual se hace más preciso y hace más daño mientras más tiempo se cargue. Tiene la apariencia de un ojo flotante.

Escudero Llamada *Shielder* en el código. Su habilidad es usar un escudo que se interpone entre ataques. Tiene una apariencia parecida a la de un casco de policia.

Recolector Llamada *Collector* en el código. Su habilidad es poder recolectar frutas de arbustos y lanzarlas. Tiene una apariencia de robot con una cesta para frutas.

Apoyo Llamada Supporter en el código. Su habilidad es generar un área en el que toda entidad va mas lento. Tiene una apariencia de robot levitante.

Criatura Llamada *Creature* en el código. Es la única con 2 habilidades y ambas son ofensivas. Pueden atacar con una embestida con la que son capaces de consumir frutas para restaurar salud. Su otra habilidad es un disparo de largo alcance. Tiene una apariencia de híbrido entre rana y seta.

5.2 *Gameplay*

El objetivo del juego es encontrar la asignación correcta de Frutas en Bowls, mecánica inspirada en el juego MasterMind. Las frutas pueden ser recogidas por el recolector en las plantas en las que crecen, pero también pueden ser comidas por las criaturas para recuperar salud.

Una vez recogidas las frutas, pueden ser lanzadas, y de tocar algún Bowl, se quedará guardada. Cuando todos los 3 Bowls tengan alguna fruta guardada, se indicará el número de frutas puestas en el bowl correcto, y el número de frutas puestas en el bowl incorrecto.

El juego termina cuando los 3 Bowls tengan asignadas las frutas correspondientes.

Los 2 jugadores tienen un grupo de 4 personajes, cada uno con habilidades diferentes, y que pueden controlar manualmente en cualquier momento, esto se hace mediante la ruleta de selección:

Cada habilidad tiene una forma diferentes de funcionar:

Ataque de Cazador Manteniendo el botón de ataque se carga el disparo, y al soltar se dispara. Si no se carga al máximo el disparo tiene una desviación, y el daño no es el máximo. Después de disparar no se puede usar el ataque por un breve periodo de tiempo. Al mantener el botón se mantiene también la dirección a la que se mira.

Defensa del Escudero Manteniendo el botón de ataque se hace aparecer un escudo en la dirección en la que mira. el escudo no aparece inmediatamente, sino después de un pequeño intervalo de tiempo. Al usar la habilidad se mantiene la dirección a la que se mira.

Habilidades del recolector Tiene dos habilidades dependiendo si lleva o no una fruta. Si no lleva una fruta, puede recogerlas de plantas, colocandose cerca de una planta, mirando en su dirección y manteniendo el botón de ataque por un tiempo. Si lleva una fruta, solo presionando el botón de ataque se lanzará la fruta. Si el recolector recibe daño, perderá la fruta que llevaba.

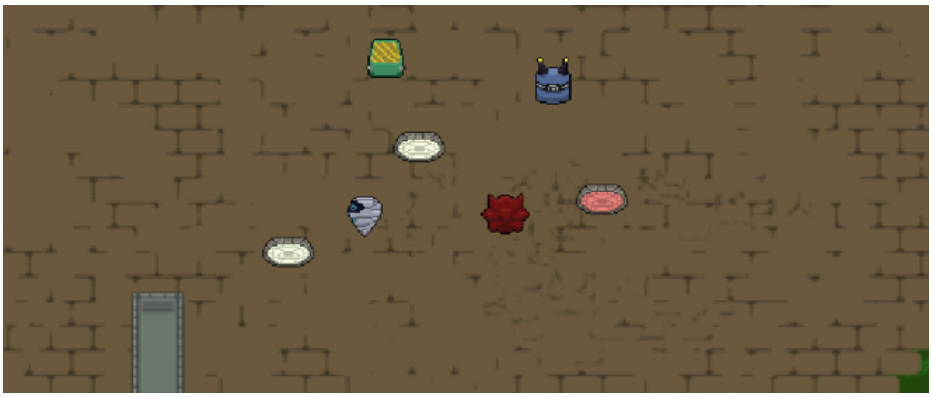
Aura de Apoyo Manteniendo presionado el botón de ataque se generará un aura que ralentiza a cuaquiera que toque el aura. Hay un pequeño retraso tanto para generar el aura como para desactivarla. Solo se puede usar por un tiempo, al agotarse el tiempo de uso se tendrá que esperar a que recargue.

Habilidades de la Criatura Para usar las habilidades solo hace falta presionar los botones correspondientes. El botón de ataque sirve para usar la embestida, hace daño a quien esté en frente y sirve también para consumir frutas y recuperar vida. El segundo botón, que el jugador usa para cambiar de personaje, sirve para usar un disparo. Después de usar cualquier ataque hay un breve periodo de tiempo en el que no se puede atacar.

Si alguno de los personajes del jugador se queda sin puntos de salud, se quedará fuera de juego, sin poder reaccionar. Luego de un tiempo volverán al juego

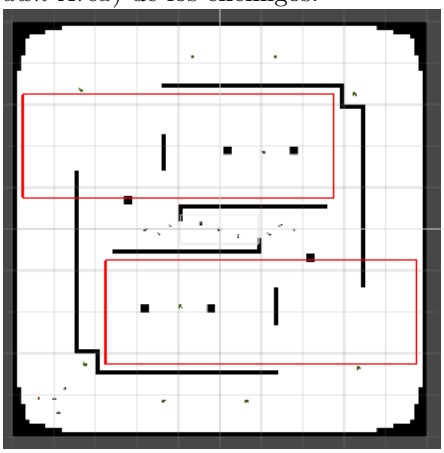
5.3 Diseño del Entorno

El mapa está distribuido de manera simétrica, donde la sección superior derecha y la inferior izquierda son iguales. En el centro del mapa se encuentran los Bowls. A la entrada de este centro hay placas que se colorean según el número de frutas puestas correcta (verde) e incorrectamente (rojo).



Por todo el mapa hay plantas de las que crecen frutas, las cuales deben ser llevadas a algún Bowl del centro.

Los jugadores empiezan en un área segura, apartados del área de aparición (*Spawn Area*) de los enemigos.



En la zonas rojas se crean enemigos, los jugadores empiezan en la esquina inferior y izquierda y superior derecha.

5.4 Arte

Todo el arte del videojuego esta inspirado en el estilo del *Pixel Art*.Cada *sprite* de las entidades estan hechas a una resolución de 32x32 píxeles. No hay ningún límite en cuanto a la paleta de colores que se usa.

La perspectiva del videojuego es *top-down*, es decir, se simula una perspectiva vista desde arriba hacia abajo.

5.5 Interfaz de Usuario

El videojuego esta diseñado para necesitar solo dos ejes direccionales, equivalente a 4 botones, más dos botones extras para acciones específicas.

Los botones direccionales sirven para mover al personaje correspondiente. Para los jugadores, uno de los botones extras se usa para usar la habilidad y otro para cambiar de personaje.

Aunque los *NPCs* no sean controlados por personas, virtualmente tienen dos botones que son usados para sus dos habilidades (Botón 1 para Embestir y Botón 2 para Disparar).

Capítulo 6

Implementación

6.1 Estructura

Unity esta desarrollado para ser trabajado con lo que se conoce como una arquitectura basada en componentes. En Unity, todo lo que se muestra al jugador esta dentro de una escena (*Scene*) y se les dice *GameObjects*, desde las entidades que controla el jugador hasta las paredes y los entidades que se encargan de instanciar otras entidades. Estos *GameObjects* tienen un comportamiento descrito por sus componentes. Existen componentes con un comportamiento predeterminado, pero también se pueden hacer componentes propios. Esto se hace mediante *Scripting*, usando objetos que derivan del objeto *Monobehaviour*.

6.2 Input

Debido a que el jugador puede cambiar de entidad y que todas las entidades pueden ser controladas por IA, es conveniente hacer un script que maneje el control de las entidades.

La clase que se encarga de esto es el *InputController* y es un componente que tienen todas las entidades del juego y que sirve para que los scripts respondan tanto a el jugador como a IAs.

```
public InputConfig m_input;

private bool up;
private bool previous_up;

private bool down;
private bool previous_down;

private bool left;
private bool previous_left;

private bool right;
private bool previous_right;

private bool b1;
private bool previous_b1;

private bool b2;
private bool previous_b2;
```

La variable m_input es la que usa el script para responder al input del jugador, de no haber esta variable (caso de ser *null*) se usan las otras variables. Esta variable es un objeto especial, es del tipo *InputConfig* que es un script propio que deriva de *ScriptableObjects*, una clase de Unity. De entre las ventajas que tienen los *ScriptableObjects*, se ha visto conveniente usarlo ya que mantienen los datos independientemente de la ejecución de las escenas. Esto significa que, de querer desarrollarse un menú para cambiar los inputs, no haría falta un script que mantenga los datos a través de las escenas.

La clase *InputConfig* es una clase pequeña que guarda las teclas que corresponden a cada botón y métodos para sabes si se han presionado.

Cada variable corresponde a si un botón esta presionado o no, y si en el frame anterior ha estado presionado.

Esta clase tiene dos tipos de funciones: los que devuelven valores del input, y los que sirven para cambiar el input. El formato de los nombres del primer tipo de funciones es:

Get(Nombre del Botón)[Pressed—Released]()

Las versiones *Pressed* y *Released* devuelven si la llamada corresponde a la frame en la que han sido presionadas o soltadas respectivamente.

Las funciones estan escritas todas de esta manera:

```
public bool GetDown()
{
    if (m_input != null)
    {
        return m_input.GetDown();
    }
    return down;
}

public bool GetDownPressed()
{
    if (m_input != null)
    {
        return m_input.GetDownPressed();
    }
    return down && !previous_down;
}

public bool GetDownReleased()
{
    if (m_input != null)
    {
        return m_input.GetDownReleased();
    }
    return !down && previous_down;
}
```

El segundo tipo de función son más sencillas:

```
public void SetUp(bool press)
{
    up = press;
}
public void SetDown(bool press)
{
    down = press;
}
public void SetLeft(bool press)
{
    left = press;
}
public void SetRight(bool press)
{
    right = press;
}
public void SetButton1(bool press)
{
    b1 = press;
}
public void SetButton2(bool press)
{
    b2 = press;
}
```

Las variables que guardan el valor de los botones del frame anterior se debe hacer después de que se hayan usado los valores del frame. Es decir, los valores del frame anterior se tienen que conservar hasta que ningún script vaya a usar el input. Para esto se usa un tipo especial de actualización llamado *LateUpdate*, que se llama luego de los *Update* de todos los componentes.

6.3 Lógica del Juego

Aqui se explicarán los scripts que tienen que ver con como funcionan los objetos de las escenas.

6.3.1 PlayerController

Esta clase la tienen todos los *GameObjects* que son entidades que "juegan" en la escena. Se encarga de el movimiento y la orientación de la entidad .

Las variables de este script son las siguientes:

```
public Direction facing;
public bool inertia;
public bool moveable;
public bool facelocked;
public bool Out;
public float speedModifier;
public float speedOffset;
public float Speed;
public Vector2 movementVector = new Vector2(0,0);

public Rigidbody2D m_rb;
public InputController m_input;
public Damageable m_d;
public Animator m_anim;
```

La enumeración *facing* indica la orientación de a entidad.

El m_input guarda una referencia al *InputController* de la entidad, e indica a que input responde.

Las variables *inertia, moveable* y *facelocked* son booleanos que guardan si se debe actualizar el movimiento y la orientación.

Las variables *speed,speedModifier* y *speedOffset* son números con los que se calcula la velocidad a la que se mueve la entidad.

MovementVector es un vector que guarda hacia donde se debe mover la entidad.

El resto de variables son referencias a otros componentes.

Como todos los componentes, este script actua en los *Updates*:

```
// Update is called once per frame
void Update()
{
    if (!Out)
    {
        if (!facelocked) ChangeFacing(ref facing);
        if(inertia) movementVector = new Vector2(0,0);
        if (moveable) movementVector = CalculateDir();
    }

}

void FixedUpdate()
{
    //Vector2 scalarBoost = movementVector.normalized *speedOffset;
    if(inertia)m_rb.velocity = movementVector*(Speed+speedOffset)*speedModifier;
}
```

Tiene dos Updates diferentes, en *Update* se actualiza el movimiento y la orientación según el input, y en *FixedUpdate*, se mueve la entidad.

Para esto se usan dos funciones que calculan el movimiento y la orientación:

```
public Vector2 CalculateDir()
{
    int up = (m_input.GetUp() ? 1 : 0);
    int down = (m_input.GetDown() ? -1 : 0);
    int left = (m_input.GetLeft() ? -1 : 0);
    int right = (m_input.GetRight() ? 1 : 0);
    Vector2 dir = new Vector2((right + left), (up + down)).normalized;
    m_anim.SetBool("Moving", dir.magnitude > 0);
    return dir;
}
```

El movimiento de la entidad se calcula según las teclas direccionales que esten presionadas. Se suman las direcciones que corresponden a cada tecla presionada y luego se normaliza.

```
private void ChangeFacing(ref Direction face)
{
    int up = (m_input.GetUp() ? 1 : 0);
    int down = (m_input.GetDown() ? -1 : 0);
    int left = (m_input.GetLeft() ? -1 : 0);
    int right = (m_input.GetRight() ? 1 : 0);
    Vector2 dir = new Vector2((right + left), (up + down));

    if ((dir.x > 0 && face == Direction.LEFT) || dir.x > Mathf.Abs(dir.y))
    {
        face = Direction.RIGHT;
    }
    else if ((dir.x < 0 && face == Direction.RIGHT) || dir.x < -Mathf.Abs(dir.y))
    {
        face = Direction.LEFT;
    }
    else if ((dir.y > 0 && face == Direction.DOWN) || dir.y > Mathf.Abs(dir.x))
    {
        face = Direction.UP;
    }
    else if ((dir.y < 0 && face == Direction.UP) || dir.y < -Mathf.Abs(dir.x))
    {
        face = Direction.DOWN;
    }

    if(m_anim != null) m_anim.SetInteger("Facing", (int) face);
}
```

De forma similar a como se calcula el movimiento, se calcula la dirección que corresponde a las teclas presionadas. La idea de calcular la orientación como se hace aquí es dar prioridad a la primera orientación asignada al empezar a moverse y no a un eje en específico.

Finalmente, debido a las mecánicas del juego, cuando la entidad esta fuera de juego y cuando vuelve al juego, deben actualizarse ciertas variables, como *moveable* por ejemplo. Esto se hace usando *delegates*:

```
private void OnEnable()
{
    if (m_d != null)
    {
        m_d.DeathEvent += PlayerDeath;
        m_d.WakeUpEvent += PlayerWakeUp;
    }
}

private void OnDisable()
{
    if (m_d != null)
    {
        m_d.DeathEvent -= PlayerDeath;
        m_d.WakeUpEvent -= PlayerWakeUp;
    }
}

void PlayerWakeUp()
{
    facelocked = false;
    moveable = true;
    Out = false;
}

void PlayerDeath()
{
    facelocked = true;
    moveable = false;
    Out = true;
}
```

Esto hace que las funciones *PlayerWakeUp* y *PlayerDeath* cuando en el componente Damageable se llamen los eventos *WakeUpEvent* y *DeathEvent* respectivamente, los cuales se llaman cuando la entidad se restaura y cuando su salud baja a 0 respectivamente.

6.3.2 HunterController

Este script se encarga de manejar la habilidad del Cazador.

```
public float aimingtime;
public float startingprecision;
public float shootingspeed;
public float power;
private bool shootReady;

[SerializeField]
private float cooldown;

private Coroutine abilityRoutine = null;
public PlayerController m_pc;
public GameObject bullet;
public InputController m_input;
```

Las variables *aimingtime* y *startingprecision* indican cuanto tiempo debe cargar el disparo, y con cuanto tiempo de ventaja empieza.

Las variables *shootingspeed* y *power* son los parámetros para el disparo. Indican la velocidad del disparo y el daño que hace.

La variable shootReady indica cuando el disparo está cargado al máximo.

La variable *cooldown* indica cuánto tiempo se espera después de poder volver a usar la habilidad.

Se guarda también una referencia a la co-rutina que se usa para la habilidad.

Se guarda un *GameObject* del disparo que usa la habilidad.

La parte del código que se encarga de la habilidad esta en el *Update*:

```
void Update () {
    if (!m_pc.Out)
    {
        if (m_input.GetButton1Pressed()) Skill();
    }
}

private void Skill()
{
    if (abilityRoutine == null) abilityRoutine = StartCoroutine("Charge");
}

private IEnumerator Charge()
{
    float timeaiming = startingprecision;
    m_pc.facelocked = true;

    while (m_input.GetButton1())
    {
        timeaiming += Time.deltaTime;
        if (timeaiming > aimingtime)
        {
            timeaiming = aimingtime;
            shootReady = true;
        }
            //Debug.Log(timeaiming);
        yield return null;
    }
    Shoot(timeaiming);
    float timeElapsed = 0;
    while (timeElapsed < cooldown)
    {
        timeElapsed += Time.deltaTime;
        yield return null;
    }
    m_pc.facelocked = false;
    abilityRoutine = null;
}
```

Se comprueba si se ha presionado el botón de la habilidad. Y entonces si no se ha iniciado ninguna rutina se inicia la rutina *Charge*.

La primera parte de la corutina es inicializar el tiempo que lleva cargando a *startingprecision* y se bloquea la orientación de la entidad. Entonces cada *frame* se calcula cuanto tiempo va cargando mientras se mantenga el botón de la habilidad. Cuando se suelte el botón se dispara y entonces se espera el tiempo hasta poder volver a usar la habilidad.

La función para el disparo es la siguiente:

```
private void Shoot(float aim)
{

    float chargeRate = aim / aimingtime;
    float deviation = Random.Range(-1f + chargeRate, 1f - chargeRate);
    Vector2 shootingVector = new Vector2(0, 0);
    switch (m_pc.facing)
    {
        case Direction.UP:
            shootingVector += new Vector2(deviation, 1);
            break;
        case Direction.DOWN:
            shootingVector += new Vector2(deviation, -1);
            break;
        case Direction.LEFT:
            shootingVector += new Vector2(-1, deviation);
            break;
        case Direction.RIGHT:
            shootingVector += new Vector2(1, deviation);
            break;
    }
    GameObject shotbullet = Instantiate(bullet, transform.position + new Vector3(shootingVector.x, shootingVector.y), Quaternion.identity) as GameObje
    BulletScript bs = shotbullet.GetComponent<BulletScript>();
    if (bs != null)
    {
        bs.SetBullet(shootingVector * shootingspeed, power);
        bs.SetOffset((1-chargeRate)*power);
    }
    shootReady = false;
}
```

Primero se calcula el porcentaje de cuanto se ha cargado y la dirección en la que irá el proyectil. A la dirección se le suma una desviación dependiendo de cuanto se haya cargado. Finalmente se instancia el proyectil y se asignan la dirección y el daño que hace el proyectil.

Este script también tiene funciones que se llaman en momentos específicos:

```
private void OnEnable()
{
    if (m_pc.m_d != null)
    {
        m_pc.m_d.DeathEvent += HunterDeath;
    }
}

private void OnDisable()
{
    if (m_pc.m_d != null)
    {
        m_pc.m_d.DeathEvent -= HunterDeath;
    }
}

void HunterDeath()
{
    if (abilityRoutine != null)
    {
        StopCoroutine(abilityRoutine);
        abilityRoutine = null;
        shootReady = false;
    }
}
```

Esto hace que se detenga la corutina de abilidad si es que hay alguna en el momento de bajar a 0 la salud.

6.3.3 SupporterController

Este script se encarga de la habilidad del apoyo.

```
[SerializeField]
private float lag;

[SerializeField]
private float radius;

[SerializeField]
private float totalFuel;

[SerializeField]
private float fuel;

[SerializeField]
private PlayerController m_pc;

[SerializeField]
private CircleCollider2D areaOfEffect;

private Coroutine abilityRoutine;

public InputController m_input;
```

El *lag* es el tiempo que tarda entre el input y el efecto de activar o desactivar la habilidad.

La variable *radius* es el radio del área circular sobre el que tiene efecto el aura.

Las variables *fuel* y *totalFuel* indican cuanto tiempo queda para mantener la habilidad funcionando y cuanto tiene como máximo.

Se tiene referencias a otros componentes, como el *Collider* que representa el área de la habilidad, y una referencia a la co-rutina de la habilidad.

```
// Update is called once per frame
void Update () {
    if (!m_pc.Out)
    {
        if (m_input.GetButton1Pressed()) Skill();
    }
    Recover();
}

void Recover()
{
    if (abilityRoutine == null && fuel < totalFuel)
    {
        fuel += Time.deltaTime;
        if (fuel > totalFuel) fuel = totalFuel;
    }
}

void Skill()
{
    if (abilityRoutine == null) abilityRoutine = StartCoroutine("Slow");
}
```

En el *Update* se responde al input activando la co-rutina encargada de la habilidad si no hay, y recuperar tiempo disponible para usar la habilidad.

```
IEnumerator Slow()
{
    float timeElapsed = 0;
    while (timeElapsed < lag && m_input.GetButton1())
    {
        timeElapsed += Time.deltaTime;
        yield return null;
    }

    areaOfEffect.radius = radius;
    float timeUnpressed = 0;
    while (fuel > 0 && (m_input.GetButton1() || timeUnpressed < lag) )
    {
        fuel -= Time.deltaTime;
        if (!m_input.GetButton1()) timeUnpressed += Time.deltaTime;
        else timeUnpressed = 0f;

        yield return null;
    }
    if (fuel < 0) fuel = 0f;

    areaOfEffect.radius = 0.5f;
    abilityRoutine = null;
}
```

En esta co-rutina al inicio se tiene la espera debido a la variable *lag*. Se aumenta el rango del área implementado como un *CircleCollider2D* de Unity una vez se ha activado la habilidad se controla cuando se debe desactivar la habilidad, que es disminuir el radio del área del *CircleCollider2D*.

El objeto que tiene este *CircleCollider2D* también tiene un script que se encarga del efecto de la habilidad:

```
private void OnTriggerEnter2D(Collider2D collision)
{
    PlayerController c_pc = collision.gameObject.GetComponent<PlayerController>();

    if (c_pc != null)
    {
        c_pc.speedModifier /= 2;
    }
}

private void OnTriggerExit2D(Collider2D collision)
{
    PlayerController c_pc = collision.gameObject.GetComponent<PlayerController>();

    if (c_pc != null)
    {
        c_pc.speedModifier *= 2;
    }
}
```

6.3.4 CollectorController

Este script controla la habilidad del recolector. El recolector usa *BoxCollider2D* para describir el área de recolección y también se explicará en este apartado.

```
public float collectionTime;
[SerializeField]
private PlayerController m_pc;

private Coroutine collectRoutine;

private GameObject Fruit;

[SerializeField]
private float fruitVelocity;

[SerializeField]
private AreaCollectorScript UpperArea;
[SerializeField]
private AreaCollectorScript LowerArea;
[SerializeField]
private AreaCollectorScript LeftArea;
[SerializeField]
private AreaCollectorScript RightArea;

public InputController m_input;
```

La variable *collectionTime* indica cuánto tiempo hace falta usar la habilidad para recolectar una fruta.

La referencia *Fruit* es la fruta que lleva el recolector. Si no lleva ninguna el valor es *null*.

La variable *fruitVelocity* indica la velocidad de la fruta al ser lanzada.

También se tienen referencias a las áreas que se usan para la recolección, entre otras cosas.

```
void Update () {
    if (!m_pc.Out)
    {
        if (m_input.GetButton1Pressed())
        {
            if (Fruit == null) PickFruit();
            else ThrowFruit();
        }
    }
}
```

En el *Update* al usarse la habilidad, se llaman a diferentes funciones dependiendo de si el recolector lleva una fruta o no.

```
void PickFruit()
{
    AreaCollectorScript areaFacing = null;
    switch (m_pc.facing)
    {
        case Direction.RIGHT:
            areaFacing = RightArea;
            break;
        case Direction.UP:
            areaFacing = UpperArea;
            break;
        case Direction.LEFT:
            areaFacing = LeftArea;
            break;
        case Direction.DOWN:
            areaFacing = LowerArea;
            break;
    }

    if (collectRoutine == null && areaFacing.spawnerReference != null) collectRoutine = StartCoroutine("Collect", areaFacing);
}
```

Al llamarse a la función *PickFruit* primero se guarda el área a usar según la orientación de la entidad. Una vez decidido esto se mira si no se esta recolectando ya y si en ese área hay alguna planta que pueda dar frutas.

```
IEnumerator Collect(AreaCollectorScript areaFacing)
{
    m_pc.facelocked = true;
    float timeCollecting = 0f;

    while (m_input.GetButton1()
        && timeCollecting < collectionTime
        && areaFacing.spawnerReference != null
        && areaFacing.spawnerReference.grownFruit != null)
    {
        timeCollecting += Time.deltaTime;
        yield return null;
    }

    if (timeCollecting >= collectionTime
        && areaFacing.spawnerReference != null
        && areaFacing.spawnerReference.grownFruit != null)
    {
        Fruit = areaFacing.spawnerReference.grownFruit;
        areaFacing.spawnerReference.grownFruit = null;
        Fruit.transform.position = new Vector3 (gameObject.transform.position.x, gameObject.transform.position.y + 0.15f, -0.2f);
        Fruit.transform.parent = gameObject.transform;
    }

    m_pc.facelocked = false;
    collectRoutine = null;
}
```

Primero, se bloquea la orientación y, mientras se este usando la habilidad, mientras el tiempo usando la habilidad no llegue al tiempo de recolección y mientras haya una fruta que recoger, se aumenta el tiempo que se lleva recolectando.

Luego si se ha llegado al tiempo de recolección y hay una fruta que recoger, se guarda la fruta en la referencia *fruit* y se elimina la referencia de la planta, el *GameObject* de la fruta se vuelve hijo del *GameObject* del recolector para que al llevarlo se mueva con el personaje. Finalmente se desbloquea la orientación y se limpia la referencia de la co-rutina.

```
void ThrowFruit()
{
    Fruit.transform.parent = null;

    Vector2 throwingDirection = new Vector2(0, 0);
    switch (m_pc.facing)
    {
        case Direction.RIGHT:
            throwingDirection.x += 1;
            break;
        case Direction.UP:
            throwingDirection.y += 1;
            break;
        case Direction.LEFT:
            throwingDirection.x -= 1;
            break;
        case Direction.DOWN:
            throwingDirection.y -= 1;
            break;
    }

    Vector3 offset = throwingDirection;
    Fruit.transform.position = gameObject.transform.position  + offset;

    Rigidbody2D fruitRB = Fruit.AddComponent<Rigidbody2D>();
    fruitRB.gravityScale = 0;
    fruitRB.freezeRotation = true;
    fruitRB.velocity = throwingDirection * fruitVelocity;
    FruitScript fs = Fruit.GetComponent<FruitScript>();

    if (fs != null) fs.destroyOnCollision = true;

    BoxCollider2D bc = Fruit.GetComponent<BoxCollider2D>();

    if (bc != null) bc.isTrigger = false;


    Destroy(Fruit, 5f);
    Fruit = null;
}
```

Se desvincula la fruta del recolector para que se pueda mover por cuenta propia. Se calcula la dirección hacia la que se lanza la fruta en función de la orientación, se le añade al *GameObject* de la fruta un *Rigidbody2D*, un componente de Unity, para poder moverlo y se configuran el script de las Frutas. Se ejecuta una función para que la fruta se destruya en 5 segundos y se limpia la referencia de *fruit*.

```
private void OnEnable()
{
    if (m_pc.m_d != null)
    {
        m_pc.m_d.HurtEvent += CollectorHurt;
    }
}

private void OnDisable()
{
    if (m_pc.m_d != null)
    {
        m_pc.m_d.HurtEvent -= CollectorHurt;
    }
}

void CollectorHurt()
{
    if (collectRoutine != null)
    {
        StopCoroutine(collectRoutine);
        collectRoutine = null;
    }
    if (Fruit != null)
    {
        Destroy(Fruit);
        Fruit = null;
    }
}
```

Debido al comportamiento del recolector, usa unas funciones que no usan el resto. *HurtEvent* se llama cada vez que una entidad es dañada. Esto quiere decir que cada vez que el recolector es dañado, deja de recoger, y de tener una fruta, el recolector la pierde.

```
public GameObject PlantReference;
public FruitSpawnerScript spawnerReference;

private void Awake()
{
    PlantReference = null;
}

private void OnTriggerEnter2D(Collider2D collision)
{
    FruitSpawnerScript spawner = collision.gameObject.GetComponent<FruitSpawnerScript>();
    if (spawner != null)
    {
            PlantReference = collision.gameObject;
            spawnerReference = spawner;
    }
}

private void OnTriggerExit2D(Collider2D collision)
{
    GameObject parent = collision.gameObject;
    if (parent == PlantReference)
    {
        PlantReference = null;
        spawnerReference = null;
    }
}
```

Finalmente, cada área del recolector tiene este script, que simplemente guarda referencias de las plantas que entran en este área, y las elimina al salir.

6.3.5 ShielderController

El script del escudero tiene las variables:

```
public PlayerController m_pc;
private float lag;

GameObject activeShield;

public GameObject UpperShield;
public GameObject LowerShield;
public GameObject RightShield;
public GameObject LeftShield;

public InputController m_input;

Coroutine shieldRoutine;
```

La variable *lag* indica cuanto tarda el escudo en aparecer. También se guarda referencias, entre otros componentes, a los *GameObject* que representan a los escudos, y al escudo activo.

```
void Update () {
    if (!m_pc.Out) {
      if(m_input.GetButton1Pressed()) Skill();
    }
}

void Skill()
{
    if (shieldRoutine == null)
    {
        shieldRoutine = StartCoroutine("ShieldRoutine");
    }
}

IEnumerator ShieldRoutine()
{
    m_pc.facelocked = true;
    float charging = 0f;
    while (m_input.GetButton1() && charging < lag)
    {
        charging += Time.deltaTime;
        yield return null;
    }
    if (charging >= lag) {
        ShieldUp();
        while (m_input.GetButton1()) yield return null;
        ShieldDown();
    }
    m_pc.facelocked = false;
    shieldRoutine = null;
}
```

Si se presiona el botón de habilidad se usa la rutina *shieldRoutine* para usar el escudo. Al inicio se bloquea la orientación del personaje, se espera hasta que pase el tiempo indicado por *lag*, una vez pasado este tiempo se activa el escudo con *ShieldUp* hasta que se deja de presionar el botón de habilidad, entonces se llama a *ShieldDown* para desactivar el escudo activo. Se desbloquea la orientación y se limpia a referencia de la co-rutina.

```
void ShieldUp()
{
    switch (m_pc.facing)
    {
        case Direction.UP:
            activeShield = UpperShield;
            break;
        case Direction.DOWN:
            activeShield = LowerShield;
            break;
        case Direction.LEFT:
            activeShield = LeftShield;
            break;
        case Direction.RIGHT:
            activeShield = RightShield;
            break;
    }
    activeShield.SetActive(true);
}
void ShieldDown()
{
    activeShield.SetActive(false);
    activeShield = null;
}
```

Las funciones *ShieldUp* y *ShieldDown* son estas. La primera activa el escudo que corresponde a la orientación y la segunda lo desactiva.

6.3.6 CreatureController

```
[SerializeField]
private PlayerController m_pc;

[SerializeField]
private Animator m_anim;

[SerializeField]
private float cooldown;

[SerializeField]
private float biteSpeed;

[SerializeField]
private float timeBiting;

[SerializeField]
private GameObject UpperHitbox;
[SerializeField]
private GameObject LowerHitbox;
[SerializeField]
private GameObject LeftHitbox;
[SerializeField]
private GameObject RightHitbox;

[SerializeField]
private GameObject spitBullet;

[SerializeField]
private float bulletSpeed;

[SerializeField]
private float bulletPower;

private Coroutine abilityRoutine;

public InputController m_input;
```

La variable *cooldown* indica el tiempo que tarda el personaje en poder volver a usar un ataque.

Las variables *biteSpeed* y *timeBiting* indican el la velocidad y el tiempo de la embestida.

Las variables *bulletSpeed* y *bulletPower* indican la velocidad y el daño que hace el proyectil disparado.

También se tienen referencias a las hitbox que hacen daño al embestir.

```
void Update () {
    if (!m_pc.Out)
    {
        if (m_input.GetButton1Pressed()) Skill("Bite");
        if (m_input.GetButton2Pressed()) Skill("Spit");
    }
}

void Skill(string ability)
{
    if (abilityRoutine == null) abilityRoutine = StartCoroutine(ability);
}
```

Al usar una habilidad se llama a la co-rutina correspondiente.

```
IEnumerator Bite()
{
    m_pc.moveable = false; m_pc.facelocked = true;  m_pc.inertia = false;
    Vector2 biteDirection = new Vector2(0,0);
    GameObject activeHitbox = null;
    m_anim.SetTrigger("Attack");
    switch (m_pc.facing)
    {
        case Direction.UP:
            biteDirection.y = 1;
            activeHitbox = UpperHitbox;
            break;
        case Direction.DOWN:
            biteDirection.y = -1;
            activeHitbox = LowerHitbox;
            break;
        case Direction.LEFT:
            biteDirection.x = -1;
            activeHitbox = LeftHitbox;
            break;
        case Direction.RIGHT:
            biteDirection.x = 1;
            activeHitbox = RightHitbox;
            break;
    }

    activeHitbox.SetActive(true);
    Rigidbody2D body = m_pc.m_rb; body.velocity = biteDirection*biteSpeed;
    float timeElapsed = 0;
    while (timeElapsed < timeBiting)
    {
        timeElapsed += Time.deltaTime;
        yield return null;
    }

    m_pc.facelocked = false; m_pc.moveable = true; m_pc.inertia = true;
    activeHitbox.SetActive(false);
    timeElapsed = 0;
    while (timeElapsed < cooldown)
    {
        timeElapsed += Time.deltaTime;
        yield return null;
    }
    abilityRoutine = null;
}
```

Primero se bloquean el input, la velocidad y la actualización de la velocidad del *Rigidbody2D* del personaje mediante las variables del *PlayerController*. Luego según la orientación, se guarda la dirección en la que tiene que embestir y la *Hitbox* correspondiente. Entonces se activa la *Hitbox* y se asigna la velocidad de la embestida al personaje mediante el *Rigidbody2D*. Una vez pasado el tiempo de la embestida, se desactiva la hitbox y se vuelve a la normalidad la actualización en el *PlayerController*. Finalmente se espera el tiempo indicado por *cooldown* y se termina la co-rutina.

```
IEnumerator Spit()
{
    Vector2 spitDirection = new Vector2(0, 0);
    m_anim.SetTrigger("Shoot");
    switch (m_pc.facing)
    {
        case Direction.UP:
            spitDirection.y = 1;
            break;
        case Direction.DOWN:
            spitDirection.y = -1;
            break;
        case Direction.LEFT:
            spitDirection.x = -1;
            break;
        case Direction.RIGHT:
            spitDirection.x = 1;
            break;
    }
    GameObject bullet = Instantiate(spitBullet,transform.position + new Vector3 (spitDirection.x, spitDirection.y), Quaternion.identity) as GameObject;

    BulletScript bs = bullet.GetComponent<BulletScript>();
    bs.SetBullet(spitDirection * bulletSpeed, bulletPower);

    float timeElapsed = 0;
    m_pc.moveable = false;
    m_pc.facelocked = true;
    while (timeElapsed < timeBiting)
    {
        timeElapsed += Time.deltaTime;
        yield return null;
    }
    m_pc.moveable = true;
    m_pc.facelocked = false;
    timeElapsed = 0;
    while (timeElapsed < cooldown)
    {
        timeElapsed += Time.deltaTime;
        yield return null;
    }
    abilityRoutine = null;
}
```

Se calcula la dirección en la que irá el proyectil. Se instancia el proyectil en la posición del personaje con un desplazamiento para que no colisione con el propio personaje.

Se asigna el daño que hará el proyectil y la velocidad. Luego se espera un tiempo en el que el personaje no se mueve y luego otro tiempo que es la espera a volver a usar otro ataque.

```
[SerializeField]
private float damage;

[SerializeField]
private float lowOffset;

private Damageable m_d;

private void Awake()
{
    damage = 20;
    lowOffset = 10;
}

private void Start()
{
    m_d = this.GetComponentInParent<Damageable>();
}

private void OnTriggerEnter2D(Collider2D collision)
{
    Damageable damaged = collision.gameObject.GetComponent<Damageable>();
    if (damaged != null)
    {
        damaged.Damage(Random.Range((damage - lowOffset) < 0 ? 0 : (damage - lowOffset), damage));
    }
    FruitScript fs = collision.gameObject.GetComponent<FruitScript>();
    if (fs != null)
    {
        m_d.Heal(20f);
        Destroy(collision.gameObject);
    }

}
```

Finalmente, cada *Hitbox* de la criatura tiene este script, el cual se encarga de hacer daño a todo *GameObject* que entre en el área si tiene el script *Damageable* y recuperar vida si toca alguna fruta. En las variables que se ven y como es calculado el daño, se escoge un valor aleatorio como máximo *damage* y como mínimo *damage* - *lowOffset*.

6.3.7 CreatureSpawner

Este script se encarga de inicializar las instancias de criaturas en todo el mapa.

```
[SerializeField]
private List<Vector2> centers;

[SerializeField]
private List<Vector2> sizes;

[SerializeField]
private GameObject creaturePrefab;

[SerializeField]
private int quantity;

// Use this for initialization
void Start () {
    centers.Clear();
    sizes.Clear();

    centers.Add(new Vector2(-10,20));
    centers.Add(new Vector2(10, -20));

    sizes.Add(new Vector2(75,25));
    sizes.Add(new Vector2(75,25));

    for (int i = 0; i < quantity; i++)
    {
        SpawnCreature();
    }
}
```

Este script usa una lista de centros y tamaños para definir áreas en las que aparecen las criaturas. También guarda el *GameObject* a instanciar y de cuantas instancias deberían existir a a vez en el mundo.

```
void SpawnCreature()
{
    int box=Random.Range(0, centers.Count);

    Vector2 spawnPos = new Vector2(Random.Range(centers[box].x - sizes[box].x/2, centers[box].x + sizes[box].x/2),
                                   Random.Range(centers[box].y - sizes[box].y/2, centers[box].y + sizes[box].y/2));

    while (!CanSpawnHere(spawnPos))
    {
        spawnPos = new Vector2(Random.Range(centers[box].x - sizes[box].x / 2, centers[box].x + sizes[box].x / 2),
                               Random.Range(centers[box].y - sizes[box].y / 2, centers[box].y + sizes[box].y / 2));
    }
    GameObject c = Instantiate(creaturePrefab, spawnPos, Quaternion.identity) as GameObject;
    Damageable d = c.GetComponent<Damageable>();
    if (d != null)
    {
        d.DeathEvent += SpawnCreature;
    }
}

bool CanSpawnHere(Vector2 pos)
{
    return !Physics2D.OverlapCircle(pos,2);
}
```

Esta función es la que instancia a las criaturas en el mundo dentro de las coordenadas definidas por los centros y los tamaños. Una vez que se escoge una coordenada aleatoria valida, se instancia a la criatura y se le agrega la misma función al evento de muerte, por lo que cuando muera se instanciara otra criatura.

6.3.8 Damageable

Este script es encargado del manejo de la salud de los personajes y de los eventos de Daño(*HurtEvent*), Muerte (*DeathEvent*) y Despertar (*WakeUpEvent*), que se llaman cuando la entidad es dañada, salud reducida a 0 y cuando la entidad vuelve al juego.

```
public float maxHealth;
public float currentHealth;

public delegate void EventFunction();
public event EventFunction HurtEvent;
public event EventFunction DeathEvent;
public event EventFunction WakeUpEvent;

[SerializeField]
private bool Knockable;

[SerializeField]
private float outTime;

private bool damageable;
```

Las variables *maxHealth* y *currentHealth* indican los puntos de salud máxima y la actual.

Luego se tienen la declaración de eventos, que son en verdad contenedores de funciones que al ser invocados ejecutan a las funciones que contienen.

Luego tenemos la variable *Knockable* que indica si la salud al bajar a 0 el personaje se queda fuera de juego o desaparece. De quedar fuera de juego, la variable *outTime* indica cuánto tiempo.

La variable *damageable* indica si el personaje puede recibir daño.

```
public void Heal(float healing)
{
    currentHealth += healing;

    if (currentHealth > maxHealth)
    {
        currentHealth = maxHealth;
    }

}

public void Damage(float damage)
{
    if (damageable)
    {
        currentHealth -= damage;
        HurtEvent?.Invoke();

        if (currentHealth <= 0)
        {
            currentHealth = 0;
            DeathEvent?.Invoke();
            if (!Knockable)
            {
                Destroy(gameObject);
            }
            else
            {
                damageable = false;
                StartCoroutine("Rest");
            }
        }
    }
}
```

Estas dos funciones son encargadas de las interacciones con otros objetos y del manejo de la vida.

Heal se encarga de curar vida, aumentando la vida una cantidad y asegurandose que no pase de la salud máxima.

Damage se encarga de bajar la salud del personaje, y de llamar a los eventos *HurtEvent* al ser dañados y *DeathEvent* al tener a vida reducida a 0. Dependiendo del valor de *knockable* el personaje desaparecerá (Criatura) o se quedará fuera de juego (personajes de jugador). Si el caso es el segundo, se iniciará una co-rutina encargada de "despertar" otra vez al personaje.

```
IEnumerator Rest()
{
    float timeElapsed = 0;
    while (timeElapsed < outTime)
    {
        timeElapsed += Time.deltaTime;
        yield return null;
    }

    WakeUpEvent?.Invoke();
    currentHealth = maxHealth;
    damageable = true;
}
```

Despues de esperar el tiempo indicado por la variable *outTime* se llama al evento *WakeUpEvent* y se vuelve la salud a su máximo.

6.3.9 Proyectiles

Todos los proyectiles de este videojuego usan el script llamado BulletScript muy parecido al que usan los Hitboxes de la criatura.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    Damageable damaged = collision.gameObject.GetComponent<Damageable>();
    if (damaged != null)
    {
        damaged.Damage(Random.Range((damage - lowOffset) < 0 ? 0 : (damage - lowOffset), damage));
    }
    Destroy(gameObject);
}

public void SetBullet(Vector2 movement, float d)
{
    b_rb.velocity = movement;
    damage = d;
}
```

Tienen dos variables que guardan el daño y cuanto puede disminuir el daño, llamados *damage* y *lowOffset*. La función *OnTriggerEnter2D* es llamada cuando un objeto colisiona con el proyectil, y hace daño mediante el script *Damageable* con el objeto que colisione primero y luego se destruye.

La función *SetBullet* sirve para dar valor a la dirección en la que va el proyectil y el daño que hace.

6.3.10 Frutas

Las frutas necesitan dos scripts para funcionar: FruitScript que deriva de Monobehaviour y *FruitType* que deriva de ScriptableObject.

FruitType solo guarda dos variables: el *sprite* de la fruta y el color que representa a la fruta.

```
public FruitType type;
public bool destroyOnCollision;
[SerializeField]
private SpriteRenderer m_sp;
```

El script de las frutas tiene solo tres variables, *type* que indica el tipo de la fruta, *destroyOnCollision* para evitar que la fruta se destruya cuando no sea lanzada, y el *SpriteRenderer* de Unity usado para mostrar el sprite en el objeto *type*.

Las plantas en las que crecen las frutas tienen un propio script que se encarga de crear una fruta de no tener una.

```
public List<FruitType> spawntype;
[SerializeField]
private GameObject fruit;
public GameObject grownFruit;

[SerializeField]
private float spawnTime;
float timeGrowing;
```

La lista *spawntype* es la lista de tipos de frutas que la planta puede crear. La variable *fruit* es el Objeto que tiene que instanciar, en este caso el objeto de la fruta. En cambio, la variable *grownFruit* es la referencia a la fruta creada.

La variable *spawnTime* es el tiempo que tarda en crear la fruta, y *timeGrowing* es el tiempo que lleva en el proceso de crear la fruta.

```
void Update () {
    GrowFruit();
}

void GrowFruit(){
    if (grownFruit == null)
    {
        if (timeGrowing >= spawnTime)
        {
            SpawnFruit();
            timeGrowing = 0;
        }
        else
        {
            timeGrowing += Time.deltaTime;
        }
    }
}

void SpawnFruit() {
    if (spawntype.Count > 0)
    {
        int chosenType = Random.Range(0, spawntype.Count);
        fruit.GetComponent<FruitScript>().type = spawntype[chosenType];
        grownFruit = Instantiate(fruit, new Vector3 (transform.position.x, transform.position.y), Quaternion.identity) as GameObject;
    }
}
```

Si la planta no tiene ninguna fruta, se comenzará a crear una. Si el tiempo que lleva creando la fruta ha llegado al tiempo necesita, se crea la fruta con la función SpawnFruit. Esta función escoge un tipo de fruta aleatoriamente de la lista de tipos de frutas que puede crear y la instancia.

6.3.11 Bowls

```
public FruitType bowlType;
[SerializeField]
private bool overwritable;

public SpriteRenderer bowlSprite;

public delegate void EventFunction();
public event EventFunction FruitInBowlEvent;

private void Awake()
{
    bowlType = null;
    overwritable = false;
}


private void OnCollisionEnter2D(Collision2D collision)
{
    FruitScript fs = collision.gameObject.GetComponent<FruitScript>();

    if (fs != null)
    {
        if (overwritable || bowlType == null)
        {
            if(bowlType == null) FruitInBowlEvent();
            bowlType = fs.type;
            bowlSprite.color = bowlType.color;

        }

    }
}
```

Cada Bowl tiene este script, que guarda un tipo de fruta (*bowlType*), si se puede sobreescribir el tipo de fruta que lleva (*overwritable*), la imagen del bowl (*bowlSprite*) y la declaración del evento que se llama cuando llega una fruta al bowl estando sin ninguna fruta.

Aparte del script que tiene cada bowl, hay un objeto que se encarga de revisar la configuración y notificar el numero de frutas colocadas correcta e incorrectamente.

```
[SerializeField]
private List<FruitBowlScript> bowls;
[SerializeField]
private List<FruitType> gameTypes;
[SerializeField]
private List<FruitType> secretConfig;

private int bowlCount;

public delegate void RightWrongFunction(int right, int wrong);
public event RightWrongFunction RightWrongEvent;
```

El *BowlManager* guarda la lista de los scripts de los bowls (*bowls*), los tipos de fruta que se usan en el juego (*gameTypes*), la configuración secreta de tipos de fruta (*secretConfig*), el numero de bowls que tienen frutas y la declaración del evento que notifica cuantas frutas estan colocadas correcta e incorrectamente (*RightWrongEvent*).

```
void Start () {

    secretConfig.Clear();
    int numBowls = bowls.Count;
    for (int i = 0; i < numBowls; i++)
    {
        secretConfig.Add(gameTypes[Random.Range(0,gameTypes.Count)]);
        bowls[i].FruitInBowlEvent += IncreaseBowlCount;
    }

}

void IncreaseBowlCount()
{
    bowlCount++;
}
```

Al inicializar, se escoge una configuración aleatoriamente, cogiendo tipos de frutas de la lista *gameTypes*, y a cada Bowl se le añade una función *IncreaseBowlCount* al evento *FruitInBowlEvent*, que se usara para mantener la cuenta de cuantos bowls tienen frutas asignadas.

```
int numBowls = bowls.Count;

if (bowlCount < numBowls) return;

bowlCount = 0;

List<FruitType> typesList = new List<FruitType>();
typesList.Clear();

for (int i = 0; i < numBowls; i++)
{
    typesList.Add(bowls[i].bowlType);
}

List<FruitType> testList = new List<FruitType>(secretConfig);
```

Cada frame lo primero que hace el *BowlManager* es revisar el numero de bowls que tienen una fruta asignada. Si todos los bowls tienen ya asignada una fruta se inicializan dos listas,*typesList* que guarda los tipos de fruta de los bowls y *testList* que guarda la configuración secreta.

```
int rightPlace = 0;

for (int i = 0; i < numBowls; i++)
{
    if (typesList[i] == testList[i])
    {
        rightPlace++;
        typesList[i] = null;
        testList[i] = null;
    }
}

int wrongPlace = 0;

for (int i = 0; i < numBowls; i++)
{
    bool found = false;
    for (int j = 0; j < testList.Count && !found;j++)
    {
        if (typesList[i] != null && typesList[i] == testList[j])
        {
            found = true;
            wrongPlace++;
            typesList[i] = null;
            testList[j] = null;
        }
    }
}

RightWrongEvent(rightPlace, wrongPlace);

ResetBowls();
```

Primero se calculan las asignaciones correctas, asignando a *null* las posiciones correctamente colocadas. Para las asignaciones incorrectas para cada asignación en un bowl que no sea *null* revisar si hay alguno en alguna posición de la configuración secreta.

Finalmente llamar al evento *RightWrongEvent* con los númeroscalculados y resetear los bowls.

6.3.12 Cámara

La cámara solo tiene un script, y es bastante sencillo.

```
public Transform target;

// Update is called once per frame
void LateUpdate () {

        Vector2 xypos = Vector2.Lerp(transform.position, target.position, 0.1f);
        transform.position = new Vector3(xypos.x, xypos.y, transform.position.z);
}
```

Se guarda una variable *target* que es la que la cámara sigue. Se calcula una posición con Lerp, una función de Unity que calcula una interpolación lineal, es decir, un punto entre dos, usando vectores que representan coordenadas en el mundo. Finalmente se mueve la cámara a la posicion calculada.

6.3.13 TeamManager

Este script se encarga del cambio de control entre los personajes del jugador y de la UI que muestra al cambiar.

```
[SerializeField]
private InputConfig m_input;

[SerializeField]
private InputController current_control;
[SerializeField]
private InputController upControl;
[SerializeField]
private InputController downControl;
[SerializeField]
private InputController leftControl;
[SerializeField]
private InputController rightControl;

[SerializeField]
private CameraScript cameraScript;

[SerializeField]
private Image image;

[SerializeField]
private Sprite hunterSelect;

[SerializeField]
private Sprite collectorSelect;

[SerializeField]
private Sprite shielderSelect;

[SerializeField]
private Sprite supporterSelect;
```

El script guarda la la configuración del jugador al que responde (m_input). Tambien guarda los *InputController* de cada personaje del equipo, y el del que se tiene control actualmente.

Se guarda una referencia a la *CameraScript* del jugador.

También se guarda referencia al objeto que muestra la UI y las imagenes que debe mostrar en cada caso

```
private IEnumerator ChangeCharacter()
{
    current_control.m_input = null;
    while (m_input.GetButton2())
    {
        if (m_input.GetUpPressed())
        {
            current_control = upControl;
            image.sprite = collectorSelect;
            image.color = Color.white;
        }
        if (m_input.GetDownPressed())
        {
            current_control = downControl;
            image.sprite = supporterSelect;
            image.color = Color.white;
        }
        if (m_input.GetLeftPressed())
        {
            current_control = leftControl;
            image.sprite = shielderSelect;
            image.color = Color.white;
        }
        if (m_input.GetRightPressed())
        {
            current_control = rightControl;
            image.sprite = hunterSelect;
            image.color = Color.white;
        }
        yield return null;
    }
    current_control.m_input = m_input;
    cameraScript.target = current_control.gameObject.transform;
    image.sprite = null;
    image.color = Color.clear;

}
```

Al presionar el botón secundario, se iniciará una co-rutina, encargada del cambio de control. A la entidad sobre la que el jugador tiene el control actualmente se le quita la referencia del objeto que guarda la configuración del input del jugador.

Entonces, mientras se mantenga el botón secundario se responderá a las direccionales, donde dependiendo de cada dirección se cambia el control actual, el sprite de la UI y la asignación de color para hacerla visible.

Cuando se deje de presionar el botón secundario, se asigna el *InputConfig* a quien tenga el control actual, se cambia de objetivo en la cámara y se limpia la imagen de la UI.

6.4 Inteligencia Artificial

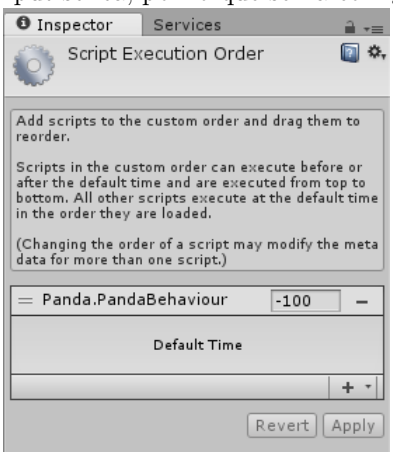
La inteligencia artificial esta implementada usando árboles de comportamiento (*Behaviour Trees*). Para esto, se usa un plugin llamado Panda BT.

Este plugin implementa una componente llamado PandaBehaviour. Este componente ejecuta un arbol de comportamiento, el cual es un archivo de texto. Las hojas del arbol de comportamiento son Tareas (llamadas *Task*) implementadas en algún *MonoBehaviour* que tenga el *GameObject*, y estas tareas son simples funciones con el atributo "[Task]". Estas tareas pueden retornar un booleano indicando si han tenido éxito o fallado, o retornar un void por lo que en cada tic del árbol no cambiara de tarea hasta que indiquemos que ha tenido

éxito o fallado.

En este proyecto, las tareas estan implementadas en dos *MonoBehaviour* diferentes, *PandaBrain* y *CreatureBrain*, para las entidades del jugador y las criaturas especificamente. Cada tipo de entidad distinta usa un árbol de comportamiento distinta, existen 5 árboles de comportamiento implementadas: CollectorBehaviour.txt para el recolector, ShielderBehaviour.txt para el escudero, HunterBehaviour.txt para el cazador, SupporterBehaviour.txt para el apoyo, y CreatureBehaviour.txt para la criatura.

Para el correcto funcionamiento de la IA, esta debe ejecutarse antes de que el input se lea, por lo que se ha configurado como se muestra a continuación:



6.4.1 PandaBrain

Este script implementa todas las tareas que necesitan las entidades que controla el jugador. Las variables que guarda este script son las siguientes:

m_input Esta variable guarda el *InputController* que usa la inteligencia para mover la entidad.

m_pc Este referencia del *PlayerController* se usa generalmente para obtener información del movimiento y la dirección de la entidad.

teammates Es una lista de *GameObjects* con los que mantiene una formación y a los que no ve como amenaza.

distanceInGroup Un número (*float*) que se usa como referencia para mantener las distancias entre el equipo, mientras más grande la distancia, más separados se mantienen.

distanceOutOfGroup Un número (*float*) que se usa como referencia para saber cuando una entidad se ha separado demasiado del grupo.

distanceAvoid Un número que se usa para detectar ataques y esquivarlos.

distanceThreatPerception Un número que indica la distancia a la que las entidades perciben alguna amenaza.

threat Una referencia al *GameObject* que se ha detectado como amenaza.

epsilon Un número (*float*) que se usa como tolerancia para hacer algunos cálculos.

target Un vector (*Vector2* de Unity) que indica la posición a la que se tiene que mover.

A continuación las funciones que se implementan:

```
Vector2 CenterOfMass()
{
    Vector2 aux = new Vector2(transform.position.x, transform.position.y);
    if (teammates.Count > 0)
    {
        aux = new Vector2(0, 0);
        foreach (GameObject mate in teammates)
        {
            aux += new Vector2(mate.transform.position.x, mate.transform.position.y);
        }
        aux /= teammates.Count;
    }
    return aux;
}

Vector2 CalculateAvoidVector()
{
    Vector2 aux = new Vector2(0,0);
    Collider2D[] hitColliders = Physics2D.OverlapCircleAll(transform.position, distanceAvoid, 1<<LayerMask.NameToLayer("Attacks"));

    for (int i = 0; i < hitColliders.Length; i++)
    {
        aux += ((Vector2) (transform.position - hitColliders[i].transform.position)).normalized;
    }

    return aux;
}
```

Estas primeras funciones no son tareas, pero si funciones que usan muchas tareas. La primera calcula el punto medio de los *GameObjects* que estan en su lista *teammates*, sin contar el propio personaje. La segunda calculala distancia contraria a los ataques que esten dentro del rango.

```
[Task]
bool InGroup()
{
    Vector2 aux = CenterOfMass();
    return Vector2.Distance(transform.position, aux) - distanceInGroup <= -epsilon;
}

[Task]
bool OutOfGroup()
{
    Vector2 aux = CenterOfMass();
    return Vector2.Distance(transform.position, aux) - distanceOutOfGroup > epsilon;
}
```

Estas son tareas que se usan para comprobar si el personaje esta lo suficientemente cerca del grupo, o si esta demasiado lejos. En ambas se calcula el centro de los y se devuelve si la distancia a ese centro es menor o mayor a los parámetros especificados (*distanceInGroup* y *distanceOutOfGroup*)

```
[Task]
void UseAbility()
{
    if (!m_input.GetButton1())
    {
        m_input.SetButton1(true);
        return;
    }
    Task.current.Succeed();
}

[Task]
void FinishAbility()
{
    if (m_input.GetButton1())
    {
        m_input.SetButton1(false);
        return;
    }
    Task.current.Succeed();
}
```

Estas son tareas que solo presionan y sueltan el botón de la habilidad. Se comprueba el estado del botón, si esta como se desea, se tiene éxito, si no, se cambia y se regresa para tener éxito en el siguiente tic.

```
void MoveToPosition(Vector2 pos)
{
    m_input.SetUp(false);
    m_input.SetDown(false);
    m_input.SetLeft(false);
    m_input.SetRight(false);

    if (pos.x - transform.position.x < -epsilon)
    {
        m_input.SetLeft(true);
    }

    if (pos.x - transform.position.x > epsilon)
    {
        m_input.SetRight(true);
    }

    if (pos.y - transform.position.y < -epsilon)
    {
        m_input.SetDown(true);
    }

    if (pos.y - transform.position.y > epsilon)
    {
        m_input.SetUp(true);
    }
}


[Task]
void MoveToTarget()
{
    MoveToPosition(target);
    Task.current.Succeed();
}
```

Estas funciones implementan el desplazamiento hacia un objetivo. El razonamiento es simple, si tengo que ir más hacia la derecha, presiono el botón de la derecha, y así para todas las direcciones. Se usa el epsilon como margen debido a que no necesariamente siempre se podrá acabar en el punto exacto, esto debido a que cada *frame* el personaje siempre se deplaza cierta distancia fija, y de quedar menos distancia entre el objetivo y el personaje que la distancia que se desplaza, al no poder desplazarse menos, se desplazara siempre pasando sobre el punto, pero nunca exactamente en el punto.

```
[Task]
void UpdateSquadDestination()
{
    target = CalculateDestinationSquad();
    Task.current.Succeed();
}


Vector2 CalculateDestinationSquad()
{
    Vector2 aux = new Vector2(transform.position.x, transform.position.y);
    //Vector2 c = CenterOfMass();
    Vector2 sMovement = SquadMovement();
    Vector2 avoid = CalculateAvoidVector();
    if (avoid.magnitude > 0)
    {
        sMovement.Normalize();
        avoid.Normalize();
    }
    aux += sMovement + avoid * 2;
    return aux;
}

Vector2 SquadMovement()
{
    Vector2 aux = new Vector2(0,0);
    Vector2 thisPosition = new Vector2(transform.position.x, transform.position.y);
    if (teammates.Count > 0)
    {
        foreach (GameObject mate in teammates)
        {
            Vector2 toMate = new Vector2(mate.transform.position.x, mate.transform.position.y) - thisPosition;
            aux += toMate - toMate.normalized * distanceInGroup;
        }
        aux /= teammates.Count;
    }
    return aux;
}
```

Esta es la implementación del movimiento grupal de los personajes que controla el jugador. La parte más interesante es el calculo del movimiento en *Squad-Movement*. Esta función garantiza que se moverán en grupo sin aglutinarse, siempre manteniendo una distancia entre ellos. Pero no garantiza que siempre se mantendrán juntos, de haber alguna pared, alguno puede quedarse atascado.

```
[Task]
void UpdateOffensiveDestination()
{
    target = CalculateOffensiveDestination();
    Task.current.Succeed();
}

Vector2 CalculateOffensiveDestination()
{
    if (threat == null) return transform.position;
    Vector2 aux = threat.transform.position;
    Vector2 c = CenterOfMass();
    Vector2 thisPosition = transform.position;
    Vector2 avoid = CalculateAvoidVector();

    switch (m_pc.facing)
    {
        case Direction.DOWN:
            aux.y = c.y;
            break;
        case Direction.UP:
            aux.y = c.y;
            break;
        case Direction.LEFT:
            aux.x = c.x;
            break;
        case Direction.RIGHT:
            aux.x = c.x;
            break;
    }
    aux -= thisPosition;
    if (avoid.magnitude > 0)
    {
        avoid.Normalize();
        aux.Normalize();
    }
    return thisPosition + aux + avoid*2;
}
```

Esta es la implementación del movimiento del cazador cuando esta atacando algún objetivo. Lo que hace es alinearse con el objetivo y el centro del resto de su equipo, lo que de ser posible moverse a esa posición, garantiza que le puede apuntar, ya que al estar alineado con el objetivo, puede disparar directamente.

```
[Task]
void UpdateSlowDestination()
{
    target = CalculateSlowDestination();
    Task.current.Succeed();
}

Vector2 CalculateSlowDestination()
{
    if (threat == null) return transform.position;
    Vector2 aux = threat.transform.position;
    Vector2 c = CenterOfMass();
    Vector2 thisPosition = transform.position;
    Vector2 avoid = CalculateAvoidVector();

    Vector2 v = (c - aux).normalized * 3;

    aux += v;

    aux -= thisPosition;

    if (avoid.magnitude > 0)
    {
        avoid.Normalize();
        aux.Normalize();
    }
    return thisPosition + aux + avoid * 2;
}
```

Este es el cálculo del movimiento del apoyo cuando esta usando su habilidad. Lo que hace es ponerse entre el centro de su equipo y la amenaza, a 3 unidades de ella.

```
[Task]
void AimThreat()
{
    if (threat == null )
    {
        Task.current.Fail(); return;
    }
    if (m_input.GetButton1())
    {
        Task.current.Fail();
    }
    Vector2 direction = threat.transform.position - transform.position;
    Direction test;
    m_input.SetUp(false); m_input.SetDown(false); m_input.SetLeft(false); m_input.SetRight(false);
    if (Mathf.Abs(direction.x) > Mathf.Abs(direction.y))
    {
        if (direction.x > 0)
        {
            test = Direction.RIGHT;
            if(m_pc.facing != test) m_input.SetRight(true);
        }
        else
        {
            test = Direction.LEFT;
            if (m_pc.facing != test) m_input.SetLeft(true);
        }
    }
    else
    {
        if (direction.y > 0)
        {
            test = Direction.UP;
            if (m_pc.facing != test) m_input.SetUp(true);
        }
        else
        {
            test = Direction.DOWN;
            if (m_pc.facing != test) m_input.SetDown(true);
        }
    }
    if (m_pc.facing == test)
    {
        if(!m_input.GetButton1())m_input.SetButton1(true);
        else Task.current.Succeed();
    }
}
```

Esta tarea sirve para apuntar al enemigo. Lo que hace es moverse en la mayor dirección de la amenaza hasta que esté orientado hacia ella. Una vez hecho esto presiona la habilidad, y tiene éxito cuando se compruebe que está presionada.

```
[Task]
bool RightAiming()
{
    if (threat == null) return false;
    Vector2 dir = threat.transform.position - transform.position;

    switch (m_pc.facing)
    {
        case Direction.DOWN:
            if (dir.y + Mathf.Abs(dir.x) < -epsilon) return true;
            break;
        case Direction.UP:
            if (dir.y - Mathf.Abs(dir.x) > epsilon) return true;
            break;
        case Direction.LEFT:
            if (dir.x + Mathf.Abs(dir.y) < -epsilon) return true;
            break;
        case Direction.RIGHT:
            if (dir.x - Mathf.Abs(dir.y) > epsilon) return true;
            break;
    }
    return false;
}
```

Con esta tarea se comprueba que la amenaza esté en la dirección en la que se está orientado, con una tolerancia de aproximadamente 45 grados.

```
[Task]
bool ValidOpportunity()
{
    if (threat == null)
    {
        return false;
    }
    else
    {
        Collider2D[] hitColliders = Physics2D.OverlapCircleAll(target, 4, 1 << LayerMask.NameToLayer("Entities"));


        for (int i = 0; i < hitColliders.Length; i++)
        {
            int team = 0;
            for (int j = 0; j < teammates.Count; j++)
            {
                if (hitColliders[i].gameObject == teammates[j].gameObject) team += 1;
            }
            if (team <=2  && hitColliders[i].gameObject != gameObject)return true;
        }
    }
    return false;
}
```

Esta tarea es usado por el apoyo para saber si en su destino hay mas de 2, es decir, el resto de su equipo, que vayan a ser afectados por su habilidad.

```
[Task]
bool NearThreat()
{
    if (threat == null)
    {
        Collider2D[] hitColliders = Physics2D.OverlapCircleAll(transform.position, distanceThreatPerception, 1 << LayerMask.NameToLayer("Entities"));

        for (int i = 0; i < hitColliders.Length && threat == null; i++)
        {
            bool team = false;
            for (int j = 0; j < teammates.Count; j++)
            {
                if (hitColliders[i].gameObject == teammates[j].gameObject) team = true;
            }
            Vector2 d = hitColliders[i].transform.position - transform.position;
            RaycastHit2D hit = Physics2D.Raycast(transform.position, d, d.magnitude, 1 << LayerMask.NameToLayer("Walls"));
            if (!team && hitColliders[i].gameObject != gameObject && !hit) threat = hitColliders[i].gameObject;
        }
        return threat != null;
    }
    else
    {
        if (Vector2.Distance(threat.transform.position, transform.position) - distanceThreatPerception > epsilon)
        {
            threat = null;
            return false;
        }
        return true;
    }
}
```

Esta tarea sirve para mantener la referencia de alguna amenaza cerca. De no haber ninguna amenaza guardada, se busca en un radio indicado por la variable *distanceThreatPerception* y de encontrar alguna entidad que no sea del equipo y que no este separada por muros se guarda como amenaza. En cambio, si hay alguna amenaza guardada se comprueba que no esté más lejos que la distancia de percepción.

Aparte de estas tareas que tienen todas las entidades con este script, también hay tareas repartidas en otros scripts como las que presento a continuación:

```
[Task]
bool Charged()
{
    return shootReady;
}

[Task]
bool ShotAvailable()
{
    return abilityRoutine == null;
}
```

Estas tareas del cazador se usan para saber si el disparo está cargado al máximo, y si ya se puede volver a usar la habilidad del disparo.

```
[Task]
bool LowOnHealth()
{
    return currentHealth < maxHealth * 0.4f;
}
```

Esta tarea en el script de *Damageable* se usa para saber si la entidad esta bajo en salud, que se considera debajo del 40% del máximo.

```
[Task]
bool EnoughFuel()
{
    return fuel > totalFuel/2;
}

[Task]
bool OutOfFuel()
{
    return fuel == 0f;
}
```

Estas tareas del apoyo sirven para saber si se tiene suficiente "combustible" para empezar a usar la habilidad, que se considera un 50% del total, y para saber cuando ya no hay.

Después de haber explicado las tareas implementadas, se explicará como han sido usadas en los árboles de comportamiento:

```
tree("Root")
    repeat
        tree ("SquadMovement")


tree("SquadMovement")
    sequence
        UpdateSquadDestination
        MoveToTarget
```

Empezando por el más sencillo, el árbol del recolector no hace mucho mas que moverse con el resto, esto es con el propósito de que cada vez que se coja alguna fruta o se lance, sea decisión del jugador que fruta coger y a donde lanzarlo. Lo que hace este árbol es repetir el árbol *SquadMovement*, y es lo que hace es una secuencia de dos tareas, calcular el movimiento en grupo y moverse.

```
tree("Root")
    repeat
        mute
            fallback
                tree ("SquadMovement")


tree("SquadMovement")
    sequence
        tree("DefendSelf")
        UpdateSquadDestination
        MoveToTarget

tree("DefendSelf")
    mute
        fallback
            sequence
                NearThreat
                AimThreat
            fallback
                sequence
                    not NearThreat
                    FinishAbility
                sequence
                    not RightAiming
                    FinishAbility
```

El segundo más simple, hace lo mismo que el recolector, excepto que este usa su habilidad para protegerse di es necesario. El extra esta en el árbol *DefendSelf* el cual detecta si hay una amenaza cerca y apunta hacia el de haberlo, si no hay ninguna o si no esta apuntando a la amenaza, deja de usar la habilidad. El nodo mute hace que el árbol siempre tenga éxito, y a pesar de que sin ese nodo igual no fallaría, evita que si al expandir el árbol se añadiese la posibilidad de fallar, como pertenece a una secuencia de *SquadMovement* no afectará en la ejecución.

También se nota desde este árbol en los siguientes el patrón de poner repeat y un mute de hijo. Esto sirve para que el árbol siempre este en ejecución así falle, independientemente de que no tenga la opción de repetir el nodo raíz.

```
tree("Root")
    repeat
        mute
            fallback
                tree ("SlowEnemies")
                tree ("SquadMovement")


tree("SquadMovement")
    sequence
        UpdateSquadDestination
        MoveToTarget

tree("SlowEnemies")
    sequence
        InGroup
        EnoughFuel
        NearThreat
        UseAbility
        mute
            repeat
                sequence
                    not OutOfGroup
                    not OutOfFuel
                    NearThreat
                    UpdateSlowDestination
                    ValidOpportunity
                    MoveToTarget
        FinishAbility
```

Siguiendo con el árbol del apoyo. Este primero comprueba si usar su habilidad o si seguir con el movimiento grupal. Usará su habilidad si esta cerca al grupo, tiene suficiente "combustible", y hay alguna amenaza cerca. Entonces se activará la habilidad y se seguirá usando mientras no se este lejos del grupo, no se quede sin "combustible", haya alguna amenaza cerca y se cumpla que no vaya a afectar al resto de su equipo.

```
tree("Root")
    repeat
        mute
            fallback
                tree ("AttackNearThreats")
                tree ("SquadMovement")


tree("SquadMovement")
    sequence
        UpdateSquadDestination
        MoveToTarget

tree("AttackNearThreats")
    sequence
        InGroup
        NearThreat
        ShotAvailable
        AimThreat
        tree("OffensiveMovement")
        FinishAbility

tree("OffensiveMovement")
    mute
        repeat
            sequence
                NearThreat
                not Charged
                not OutOfGroup
                RightAiming
                UpdateOffensiveDestination
                MoveToTarget
```

Finalmente, el árbol del cazador, muy parecido al del apoyo. En este caso usará su habilidad si está cerca del grupo, si hay una amenaza cerca y si puede atacar. De ser así, apunta al objetivo y carga el disparo. Mantendrá el disparo y moviendose en torno a la amenaza mientras la amenaza este cerca, no tenga el dispara cargado, no este fuera del grupo, y este apuntando en la dirección correcta.

6.4.2 CreatureBrain

Este script funciona muy parecido al *PandaBrain* descrito antes. Se indicarán solo las mayores diferencias.

realTarget Es un vector que indica una posición del mundo a la que se quiere llegar

fruit Similar a *threat* guarda la referencia a una fruta cercana.

mobilityRange Indica el radio en el que puede explorar la criatura.

fruitRange Indica el radio en el que puede detectar frutas cercanas.

meleeRange Indica la distancia a la que tiene que star un objeto para poder ser alcanzado con una embestida.

timeElapsed Booleano que sirve para saber si ha pasado el tiempo a esperar desde que se inició la co-rutina de esperar.

```
[Task]
bool ArrivedToTarget()
{
    return Vector2.Distance(transform.position, realTarget) <= epsilon*2;
}

[Task]
void SetRandomTarget()
{
    if (Vector2.Distance(transform.position, realTarget) <= epsilon*2)
    {
        Vector2 relativeTarget = Random.insideUnitCircle;
        float radius = Random.Range(epsilon, mobilityRange);

        RaycastHit2D hit = Physics2D.Raycast(transform.position, relativeTarget, radius, 1 << LayerMask.NameToLayer("Walls"));
        if (!hit)
        {
            realTarget = transform.position;
            realTarget += relativeTarget * radius;
        }
    }

    Vector2 avoid = CalculateAvoidVector();
    Vector2 rt = realTarget - new Vector2(transform.position.x, transform.position.y);
    if (avoid.magnitude > 0)
    {
        avoid.Normalize();
        rt.Normalize();
    }
    target = new Vector2(transform.position.x, transform.position.y) + rt + avoid;
    Task.current.Succeed();
}
```

Estas dos funciones sirven para saber si se ha llegado cerca de objetivo y para asignar un nuevo objetivo aleatorio. Como se puede ver en *SetRandomTarget* se asigna a *realTarget* un punto aleatorio dentro del radio de exploración y luego se comprueba que no hayan muros en medio, pero luego se calcula hacia donde se va a mover inmediatamente en *target*.

```
[Task]
void SetTargetNearFruit()
{
    if (fruit == null)
    {
        Task.current.Fail();
        return;
    }

    Vector2 fruitToThis = transform.position - fruit.transform.position;
    Vector2 offset = new Vector2(0,0);
    if (Mathf.Abs(fruitToThis.x) > Mathf.Abs(fruitToThis.y))
    {
        offset.x = fruitToThis.x;
    }
    else
    {
        offset.y = fruitToThis.y;
    }

    realTarget = fruit.transform.position;
    realTarget += offset.normalized * (MeleeRange - epsilon);
    Vector2 avoid = CalculateAvoidVector();
    Vector2 rt = realTarget - new Vector2(transform.position.x, transform.position.y);
    if (avoid.magnitude > 0)
    {
        avoid.Normalize();
        rt.Normalize();
    }
    target = new Vector2(transform.position.x, transform.position.y) + rt + avoid;
    Task.current.Succeed();

}
```

Esta tarea calcula cual es el punto más cercano desde el que se puede consumir una fruta cercana. Se calcula el vector de que va de la fruta objetivo a la entidad y se calcula cual es la dimension mas grande del vector, entonces se toma como posición la posición de la fruta más esa dirección calculada a una distancia indicada por *meleeRange*.

```
[Task]
bool TimeElapsed()
{
    if (elapsingTime == null) elapsingTime = StartCoroutine("ElapseTime");
    if (timeElapsed)
    {
        timeElapsed = false;
        return true;
    }
    return false;
}

IEnumerator ElapseTime()
{
    float time = 0;
    while (time < waitTime)
    {
        time += Time.deltaTime;
        yield return null;
    }
    timeElapsed = true;
    elapsingTime = null;
}
```

Esta tarea simplemente tiene éxito pasado el tiempo de haberla invocado por primera vez.

```
[Task]
bool FruitInRange()
{
    return Vector2.Distance(transform.position, fruit.transform.position) <= MeleeRange + epsilon;
}

[Task]
bool ThreatInMeleeRange()
{
    return Vector2.Distance(transform.position, threat.transform.position) <= MeleeRange + epsilon;
}
```

Este conjunto de tareas tiene éxito si la fruta o amenaza estan dentro del rango de a embestida.

Otras tareas como *AimThreat, AimFruit, AimingThreat, AimingFruit, etc* son variaciones de implementaciones ya vistas.

El script de la criatura también tiene implementada una tarea:

```
[Task]
bool AttackAvailable()
{
    return abilityRoutine == null;
}
```

Esta tarea de la criatura sirve para saber si puede volver a atacar

Para finalizar, el árbol de comportamiento de la criatura:

```
tree("Root")
    repeat
        mute
            fallback
                tree("FeedOnFruits")
                tree("DefendAgainstThreats")
                tree("Explore")

tree("Explore")
    sequence
        SetRandomTarget
        MoveToTarget
```

Esta sección es la más sencilla, parecida la parte básica de las otras entidades. Si no se alimenta de frutas ni ataca amenazas el comportamiento sera escoger un punto aleatorio y moverse hacia ese punto.

```
tree("FeedOnFruits")
    sequence
        LowOnHealth
        NearFruit
        SetTargetNearFruit
        MoveToTarget
        mute
            sequence
                FruitInRange
                fallback
                    sequence
                        AimingFruit
                        AttackAvailable
                        Attack
                        FinishAttack
                    sequence
                        AimFruit
```

Para alimentarse de frutas, se asegura que tiene la salud baja y que hay alguna fruta cerca. Entonces pone como objetivo un punto cerca de la fruta y se mueve hacia ella. Si la fruta esta en rango intentará comerla de estar apuntandola y de poder atacar, si no, intentará apuntar hacia la fruta.

```
tree("DefendAgainstThreats")
    sequence
        NearPlayers
        random(0.05, 0.95)
            repeat
                sequence
                    not TimeElapsed
                    tree("Explore")
            sequence
                fallback
                    sequence
                        ThreatInMeleeRange
                        fallback
                            sequence
                                AimingThreat
                                AttackAvailable
                                Attack
                                FinishAttack
                            sequence
                                AimThreat
                    fallback
                        sequence
                            AimingThreat
                            AttackAvailable
                            Shoot
                            FinishShoot
                        sequence
                            AimThreat
```

Para defenderse contra amenazas primero buscará si hay jugadores cerca. de ser así hay una probabilidad que explore por un tiempo, esto es para que aunque detecte amenazas se pueda mover y no solo atacar. La otra posibilidad es que si la amenaza esta en el rango de la embestida, atacar con la embestida de ser posible y si no apuntar hacia la amenaza. Si la amenaza no esta dentro del rango de la embestida, hara lo mismo pero con el disparo.

Capítulo 7

Pruebas

Antes de realizar las pruebas,se especificará el proceso que se usa para calibrar los parámetros de la IA. También se establecerán medidas que sirvan para evaluar la calidad de la inteligencia artificial. Las medidas son bastante simples, que cada árbol de comportamiento cumple el comportamiento que se le tiene encargado:

- El objetivo del árbol *SquadMovement* es mantener un movimiento grupal en el caso que no haya amenaza.
- El escudero tiene una variación del *SquadMovement* ya que tiene un nodo extra *DefendSelf*, por lo que evaluará que cumpla la función de defenderse habiendose detectado una amenaza. Lo que se hará para probar la calidad de la inteligencia artificial son enfrentamientos de 30 segundos, y la medida sera la cantidad de enfrentamientos en la que ha sido dañada la entidad, mientras menos mejor.
- En el caso del Cazador, se evaluará su efectidad usando de manera similar al escudero, se harán enfrentamientos de 1 minuto y será un punto a favor si se consigue ganar a la amenaza.
- Para el apoyo, se evaluará su efectividad usando como medida el tiempo que mantiene a amenazas dentro de su habilidad en enfrentamientos de 30 segundos.

También se considerará las habilidades de la criatura:

- La medida de explorar será la siguiente: basta con el correcto funcionamiento de escoger y moverse hacia un objetivo sin atascarse.
- Para el árbol encargado de comer frutas, simplemente las veces que come una fruta cerca cuando esta bajo de salud.
- Para el árbol de ataque, se tomará como medida las veces que golpea alguna entidad en enfrentamientos de 20 segundos.

Al haberse programado solo una estrategia de acción para cada personaje, se evaluará que los parámetros den buenos resultados.

Los escenarios escogidos varian en la cantidad y clase de entidades: el primer escenario sin enemigos, el segundo solo con una criatura, y el último escenario con todas las entidades que debe tener el producto final. La razón para hacer las pruebas de esta forma es que es mucho más fácil evaluar ciertos comportamientos en escenarios menos complejos.

Por último, especificar como se calibrarán los parámetros de la inteligencia artificial. Para comenzar, cada componente *PandaBrain* se configurará de la siguiente manera (usando de ejemplo el componente del escudero):

En el primer escenario, en el que se evalúa el movimiento, se tocarán los parametros *distanceInGroup* y *epsilon*.

En el segundo escenario se cambiará la variable *distanceAvoid* que es la que se encarga de percibir ataques para esquivarlos. Se tocarán otros parámetros pero no serán valores finales, solo se cambiarán para hacer posibles las pruebas. Por ejemplo si no cambiamos la variable *distanceThreatPerception* no detectará ninguna amenaza.

Finalmente, en el tercer y último escenario, se configurarán el resto de variables.

En todos los escenarios se informarán los parámetros que den mejores medidas.

A continuación se presentarán los resultados de cada escenario:

7.1 Escenario sin enemigos

En este escenario, se evaluará el movimiento en grupo. Primero se efectúa la calibración de los parámetros correspondientes:

- Al calibrar los parámetros, primero se escogió una distancia lo suficientemente grande para que no colisionen entre ellos.

- Se comienza a incrementar el epsilon hasta que los personajes dejen de temblar. Se deja un pequeño margen.
- Se pone a prueba que no haya temblores cuando el grupo se mueve.
- Finalmente, se escoge un *distanceInGroup* suficientemente grande para que no suelan estar en el camino de las habilidades de cada uno. Esto se para que cuando el jugador quiera usar la habilidad como la recolección, no haya ninguna entidad tapando la fruta.

Los resultados de la parametrización han sido:

- *distanceInGroup* = 3,5
- *epsilon* = 0,09

Las medidas han sido las esperadas:

- El movimiento en grupo funciona como es esperado.
- Las medidas del resto de árboles han sido las esperadas. Ningún daño, ninguna protección de ataques ni habilidad de apoyo usada.

Algunas observaciones:

- El movimiento del grupo no garantiza en ningún momento que se mantegan siempre juntos, y no era de esperar que lo hicieran. Con ciertas maniobras cerca de paredes, las entidades a veces se aislan del resto.

7.2 Escenario con solo un enemigo

Es este escenario se comprobará los resultados de los árboles correspondientes a las habilidades:

- Primero se comprobara entidad por entidad, aumentando el valor de *distanceThreatPerception*, y se medirá la eficacia de los árboles de habilidades.
- Luego se comprobará un valor bueno para *distanceAvoid*, que será el valor más pequeño que haga que ningún personaje muera en 1 minuto empezando de 0 y aumentando 0,5 la cantidad.

Las medidas de los árboles de las habilidades son las siguientes:

- Para el escudero, se hicieron 50 enfrentamientos y hubieron 37 enfrentamientos en los que no fue dañado en los 30 segundos. En ningún enfrentamiento la vida fue reducida a 0. Todos los casos en los que fue dañado fue a causa de la embestida de la criatura, ya que a corto alcance y a la velocidad que va la embestida, el escudero necesita desactivar el escudo para cambiar de dirección, y al volver a activar el escudo, debido al retraso, no reacciona al siguiente ataque.

- Para el cazador, se hicieron 50 enfrentamientos y hubieron 46 enfrentamientos en los que derroto a la criatura enfrentada. Parte de esto se debe al posicionamiento programado, que suele posicionarse cerca de sus compañeros y usarlos para distraer a la criatura.

- Para el apoyo, se hicieron 50 enfrentamientos y en promedio se mantuvo a la criatura afectada por la habilidad 18 segundos. Es lo esperado de la IA, ya que lo que se espera es 1 segundo de espera, 10 segundos de habilidad hasta agotarse, 5 segundos de recarga, 1 segundo de espera, 5 segundos de habilidad, 5 segundos de recarga, 1 segundo de espera,y 3 de habilidad, suponiendo que tiene la amenaza continúamente en rango.

Los resultados son satisfactorios, la IA funciona y hace lo que le corresponde. Se probaron diferentes valores para *distanceAvoid*: el resultado más pequeño que cumplió con la prueba fue 2.

En este escenario se hace también las pruebas con la criatura. Primero la calibración se hace de la siguiente forma:

- Primero se escoge un valor para *mobilityRange* lo suficientemente grande para que se desplace, y lo suficientemente pequeño para que no escoja posiciones mas allá de las paredes.

- Se escoge un *epsilon* la misma forma que las entidades

- Se escoge arbitrariamente un *meleeRange* que se igual o menor a la velocidad de la embestida por el tiempo de la embestida, es decir 20 * 0,2 = 4.

- Se escoge arbitrariamente el resto de de variables excepto *distanceAvoid* que se escogerá en el siguiente escenario mediante pruebas.

Los resultados del proceso anterior son:

- $mobilityRange = 7$

- $epsilon = 1{,}5$

- $meleeRange = 4$

- $FruitRange = 6$

- $distanceThreatPerception = 7$

- $waitTime = 1$

7.3 Escenario con varios enemigos y dos jugadores

Este vendría a ser el escenario del juego completo, en los que ahora las inteligencias tienen más que percibir. En este escenario quiere lograr lo siguiente:

- Obtener los parámetros restantes para las IAs
- Comprobar que no hay ningún comportamiento indeseado en un sistema más grande.

Empezando con la criatura, se prueban valores para *distanceAvoid*, que es el menor valor con el que las criaturas no se quedan atascados, y después de pruebas se escoge el valor 1, valores más pequeños también funcionan, pero se escoge el 1 por redondear y por dejar un margen.

Para los personajes del jugador, se calibrarán las variables restantes mediante pruebas.

- Para *distanceOutOfGroup* se buscará el valor mayor a *distanceInGroup* que no interrumpa la habilidad que se esté usando, pero que no aleje al personaje demasiado del grupo. Solo el cazador y el apoyo usan esta variable y como el cazador no necesita acercarse como el apoyo, el apoyo necesitará poder alejarse más del grupo. Tras pruebas, al cazador se le ha dado 5,5 y al apoyo 7.
- Para *distanceThreatPerception* se busca un valor que sea lo suficientemente grande para encontrar amenazas cercanas con las que tenga sentido usar la habilidad, pero lo suficientemente pequeñas para poder usar su habilidad sin salir del grupo y apuntar a una amenaza que sea relevante. Solo el recolector no usa esta variable. Después de pruebas, las variables para el cazador, el escudero y el apoyo son 6, 5 y 6 respectivamente.

Finalmente se hacen partidas de prueba en el que se buscan comportamientos indeseados:

- Comportamientos erráticos y poco naturales, como por ejemplo, temblores en el control de los personajes, movimiento contra las paredes.
- Falta de respuesta en las entidades, entre equipos y criaturas.

Para medir esto, de 50 partidas jugadas, esto es lo que se vio:

- A lo largo de las partidas en 34 de ellas se vio criaturas caminando contra una pared, especialmente contra los bowls. Esto es porque para comprobar si el camino entre la criatura y el objetivo esta libre se usa una línea, pero el grosor de la criatura es obviamente mayor al de la línea, por lo que no lo detecta como obstáculo. En 9 se encontraron criaturas atascadas entre ellas. No se encontró ningún temblor notorio ni poco natural.

- En todas las partidas toda entidad responde adecuadamente: el cazador es capaz de derrotar criaturas por cuenta propia, el escudero es capaz de evitar el daño de las criaturas. El apoyo probablemente necesite mas trabajo, hace lo que se esperaba pero en la práctica no funciona tan bien junto al resto del grupo; sin embargo, es considerado un problema de diseño.

También se nota que:

- Al haber más amenazas, los grupos se dispersan más, por lo que disminuir un poco la distancia que deben guardar entre ellos podría ser una medida para controlar esto.

- La sensación final del juego es la del recolector siendo el personaje principal, al tratar de llevar la fruta sin recibir daño al centro del mapa, usando el resto de personajes generalmente contra el otro jugador. Las ideas del diseño se sienten logradas en parte gracias a la IA.

Capítulo 8

Conclusiones

La técnicas de inteligencia artificial se han vuelto cada vez más sofisticadas y potentes a medida que el tiempo ha pasado. La evolución de la IA ha ido dejando una librería de métodos para afrontar diversos problemas.

Hoy en día tenemos técnicas muy potentes, pero también muy costosas, como el aprendizaje por refuerzo, las redes neuronales y los algoritmos genéticos; y estas mismas suelen opacar técnicas que una vez fueron suficientes para brindar solución a algún problema.

Cuando es sobre videojuegos, lo importante de la inteligencia artificial no es cuan potente sea, si no, que cumpla con su cometido, que satisfaga lo que el videojuego requiere. Pocos juegos actualmente necesitan tanta potencia, y requieren más bien que sean rápidos, eficaces, y poco costosos.

Pero sin duda, el estudio de nuevas técnicas y de la mejora significan más herramientas para trabajar.

En la culminación de este proyecto, hacemos un repaso de los objetivos alcanzados:

- Se ha ponderado y decidido una técnica de IA para llevar a cabo, y se ha implementado con éxito.
- Se ha tenido total libertad sobre el apartado gráfico, integrando lo que se ha considerado oportuno y necesario.
- Despues de haber hecho las pruebas, se han ajustado los parámetros de las IAs y de componentes de las entidades, para brindar una experiencia equilibrada.
- Finalmente, se han diseñado soluciones tanto a la hora de estructurar el código como a la hora de programar los comportamientos.

Capítulo 9

Trabajo Futuro

9.1 Configuración de controles

Como ha sido comentado anteriormente, lo único que haria falta es una escena en la que se puedan configurar los inputs, y que el script que configura los inputs cambia los *InputConfig* usados en el juego. Sin ningún problema de persistencia de datos entre escenas.

9.2 ML-Agents

Unity tiene una librería de *Machine Learning* que se podría usar perfectamente para la criatura. Una vez descrita solo tendria que usar el *InputController* del *GameObject* para controlar el personaje. Esto es porque cualquier componente puede controlar a los personajes mediante el *InputController*.

9.3 Frutas

Otra de las ventajas de los *ScriptableObjects* es la facilidad de expansión. Si se quiere más tipos de frutas, se hace un nuevo *FruitType* y se le asigna el sprite y el color que le representa. No hace falta hacer más. Además de que no hay que cambiar nada en cuanto a como se calculan las frutas puestas correcta e incorrectamente, al compararse funcionan como enums lo que lo hace más flexible.

9.4 Recompensa de las criaturas

Se podría sacar más provecho a los eventos de muerte de la criatura, y por ejemplo soltar alguna fruta rara o algún bonus. Esto motivaría también al jugador a interactuar con las criaturas y tendría un efecto positivo.

Bibliography

[1] Alex J. Champandard. *Top 10 Most Influential AI Games*. http://aigamedev.com/open/review/top-ai-games/, 12 de Setiembre, 2007.

[2] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, http://gameaibook.org, 2018.

[3] Richard Moss. *The history of real-time strategy games* https://arstechnica.com/gaming/2017/09/build-gather-brawl-repeat-the-history-of-real-time-strategy-games/, 9 de Setiembre, 2017.

Referencias no citadas

[4] Harbing Lou. *AI in Video Games: Toward a More Intelligent Game* http://sitn.hms.harvard.edu/flash/2017/ai-video-games-toward-intelligent-game/, 28 de Agosto, 2017.

[5] George Ferguson and James Allen *Mixed-Initiative Systems for Collaborative Problem Solving* https://www.aaai.org/ojs/index.php/aimagazine/article/view/2037/1930, AI Magazine Volume 28 Number 2 (2007).

[6] Eric Begue *Panda Behaviour Documentation* http://www.pandabehaviour.com/?page_id=23, Enero 2016.

[7] Unity Technologies *Unity Documentation* https://docs.unity3d.com/Manual/index.html, 2018.